



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ : H04L	A2	(11) International Publication Number: WO 99/14881 (43) International Publication Date: 25 March 1999 (25.03.99)																														
(21) International Application Number: PCT/US98/19316 (22) International Filing Date: 16 September 1998 (16.09.98) (30) Priority Data: <table border="0"> <tr><td>60/059,082</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,839</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,840</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,841</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,842</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,843</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,844</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,845</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,846</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> <tr><td>60/059,847</td><td>16 September 1997 (16.09.97)</td><td>US</td></tr> </table> (71) Applicant (for all designated States except US): INFORMATION RESOURCE ENGINEERING, INC. [US/US]; 8029 Corporate Drive, Baltimore, MD 21236 (US). (72) Inventors; and (75) Inventors/Applicants (for US only): KAPLAN, Michael, M. [US/US]; 4 Ocean Drive, Rockport, MA 01966 (US). DOUD, Robert, Walker [US/US]; 4 Redcoat Road, Bedford, MA 01730 (US). KAVSAN, Bronislav [US/US]; 150 Rosemont Drive, North Andover, MA 01845 (US). OBER,		60/059,082	16 September 1997 (16.09.97)	US	60/059,839	16 September 1997 (16.09.97)	US	60/059,840	16 September 1997 (16.09.97)	US	60/059,841	16 September 1997 (16.09.97)	US	60/059,842	16 September 1997 (16.09.97)	US	60/059,843	16 September 1997 (16.09.97)	US	60/059,844	16 September 1997 (16.09.97)	US	60/059,845	16 September 1997 (16.09.97)	US	60/059,846	16 September 1997 (16.09.97)	US	60/059,847	16 September 1997 (16.09.97)	US	Timothy [US/US]; 9 Birch Lane, Atkinson, NH 03811 (US). REED, Peter [US/US]; 1 Bancroft Avenue, Beverly, MA 01915 (US). (74) Agent: BODNER, Gerald, T.; Hoffmann & Baron, LLP, 350 Jericho Turnpike, Jericho, NY 11753 (US). (81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG). Published <i>Without international search report and to be republished upon receipt of that report.</i>
60/059,082	16 September 1997 (16.09.97)	US																														
60/059,839	16 September 1997 (16.09.97)	US																														
60/059,840	16 September 1997 (16.09.97)	US																														
60/059,841	16 September 1997 (16.09.97)	US																														
60/059,842	16 September 1997 (16.09.97)	US																														
60/059,843	16 September 1997 (16.09.97)	US																														
60/059,844	16 September 1997 (16.09.97)	US																														
60/059,845	16 September 1997 (16.09.97)	US																														
60/059,846	16 September 1997 (16.09.97)	US																														
60/059,847	16 September 1997 (16.09.97)	US																														
(54) Title: CRYPTOGRAPHIC CO-PROCESSOR (57) Abstract <p>A secure communication platform on an integrated circuit is a highly integrated security processor which incorporates a general purpose digital signal processor (DSP), along with a number of high performance cryptographic function elements, as well as a PCI and PCMCIA interface. The secure communications platform is integrated with an off-the-shelf DSP so that a vendor who is interested in digital signal processing could also receive built-in security functions which cooperate with the DSP. The integrated circuit includes a callable library of cryptographic commands and encryption algorithms. An encryption processor is included to perform key and data encryption, as well as a high performance hash processor and a public key accelerator.</p>																																

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon		Republic of Korea	PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

CRYPTOGRAPHIC CO-PROCESSOR

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is based on U.S. Provisional Patent Application Serial Nos. 60/059,082, 60/059,839, 60/059,840, 60/059,841, 60/059,842, 60/059,843, 60/059,844, 60/059,845 and 60/059,847, each of which was filed on September 16, 1997, the disclosures of which are incorporated herein by reference.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

Field Of The Invention

The present invention relates generally to a secure communication platform on an integrated circuit, and more particularly relates to a digital signal processor (DSP) with embedded encryption security features.

Description Of The Prior Art

Digital signal processors (DSPs) are widely used in devices such as modems, cellular telephones and facsimiles. With an increase in digital communications, data transmission security has become an issue in numerous DSP applications. A standard DSP is not capable of providing data transmission security; thus, additional hardware and software are required.

Security for digital communications is available on various integrated circuits. The integrated circuit security features include hardware implemented encryption algorithms such as the Data Encryption Standard (DES), Hash function algorithms and hardware implemented public key accelerators. The availability of this hardware makes it possible to provide security for distributed computing; however, no hardware implemented encryption algorithms have been known to be incorporated in a DSP.

Software encryption algorithms have also been developed to provide security for distributed computing. One commonly used encryption algorithm is the Data Encryption Standard (DES). DES is a block cipher which operates on 64-bit blocks of data and employs a 56-bit key. Another commonly used standard is the Digital Signature Algorithm (DSA). The DSA standard employs an irreversible public key system. These algorithms and more are part of the public domain and are available on the Internet.

Hash function algorithms are used to compute digital signatures and for other cryptographic purposes. One Hash function algorithm is the U.S. government's Secure Hash Algorithm (SHA-1).

Another security standard commonly used is the Internet Protocol Security Standard (IPsec). This standard provides security when communicating across the Internet. The standard requires DES to encrypt an Internet Protocol data packet, SHA-1 for authentication, and a public key algorithm for hand-shaking.

Since the IPsec standard requires different encryption algorithms, a software library is usually created so that a desired algorithm may be accessed when needed. Security systems employing encryption libraries are software implemented and designed specifically to run on the user's processor hardware.

Digital communication systems are not generally designed with security hardware. In most systems, security is achieved by software, such as described above, which is not entirely secure because there is no security hardware to block access to the security software by an intruder. Another problem associated with software encryption algorithms is that some of the software encryption algorithms run slower than when hardware implemented.

OBJECTS AND SUMMARY OF THE INVENTION

It is an object of the present invention to provide a digital signal processor with embedded security functions on a single integrated circuit.

It is another object of the present invention to provide a secure communications platform that can implement a user's application and dedicate cryptographic resources to encryption and decryption requests on demand.

It is another object of the present invention to provide an increase in encryption security through hardware implementations.

It is another object of the present invention to provide a security co-processor for high speed networking products such as routers, switches and hubs.

A cryptographic co-processor constructed in accordance with one form of the present invention includes a processor having encryption circuits built into it. The processor is capable of processing various applications, such as modem and networking applications. The encryption circuits and firmware make it possible to add security to the various processing applications. Hardware such as encryption and hash circuits are provided and structured to work together to provide accelerated encryption/decryption capabilities. A memory is programmed with cryptographic algorithms that support various encryption/decryption techniques. The cryptographic co-processor is structured so that a manufacturer of data communication products could substitute a current processor with the cryptographic co-processor and receive encryption capabilities with little modification to the existing product.

Since DSP's are the building block of many communication systems, a secured DSP with universal security features that may be selected by the manufacturer of the equipment in which the DSP forms part of would have far ranging benefits.

The benefits of a universal cryptographic co-processor (e.g., DSP) is that it can perform standard processor functions and standard encryption functions with no peripheral hardware or cryptographic software. Because the cryptographic co-

processor is implemented on a standard processor platform (i.e., substrate or monolithic chip), the processor that is being used in a manufacturer's product can be substituted with the cryptographic co-processor with little or no modification to the original design. The manufactured product incorporating the secure, universal co-processor now has encryption capabilities along with the original processor capabilities.

A preferred form of the cryptographic co-processor, as well as other embodiments, objects, features and advantages of this invention, will be apparent from the following detailed description of illustrated embodiments, which is to be read in connection with the accompanying drawing.

BRIEF DESCRIPTION OF THE DRAWING

Figure 1 is a block diagram of the cryptographic co-processor formed in accordance with the present invention.

Figure 1A is a block diagram similar to Figure 1 showing another view of the cryptographic co-processor of the present invention.

Figure 2 is a block diagram of the program memory preferably used in the co-processor of the present invention for MMAP=0.

Figure 3 is a block diagram of the program memory preferably used in the co-processor of the present invention for MMAP=1.

Figure 4 is a block diagram of the data memory preferably used in the co-processor of the present invention.

Figure 5 is a block diagram of the PCI memory preferably used in the co-processor of the present invention.

Figure 6 is a block diagram of the DMA subsystem preferably used in the co-processor of the present invention.

Figure 7 is a block diagram of the DSP "local" memory preferably used in the co-processor of the present invention.

Figure 8 is a flow chart showing the DMA control preferably used in the co-processor of the present invention. 4

Figure 9 is a block diagram of the hash/encrypt circuit preferably used in the co-processor of the present invention.

Figure 10 is a block diagram of the interrupt controller preferably used in the co-processor of the present invention.

Figure 11 is a block diagram of the CGX software interface preferably used in the co-processor of the present invention.

Figure 12 is a block diagram illustrating the layers of software preferably used in the co-processor of the present invention.

Figure 13 is a block diagram of the CGX overlay interface preferably used in the co-processor of the present invention.

Figure 14 is a block diagram illustrating the hierarchical interface of the CGX kernel cryptographic service preferably used in the co-processor of the present invention.

Figure 15 is a functional state diagram illustrating the CGX kernel preferably used in the co-processor of the present invention.

Figure 16 is a functional tree diagram showing the KEK hierarchy preferably used in the co-processor of the present invention.

Figure 17 is a block diagram of a symmetric key weakening algorithm preferably used in the co-processor of the present invention.

Figure 18 is a functional tree diagram showing the symmetric key hierarchy preferably used in the co-processor of the present invention.

Figure 19 is a portion of a computer program defining the PCDB data type preferably used in the co-processor of the present invention.

Figure 20 is a block diagram of the kernel block preferably used in the co-processor of the present invention.

Figure 21 is a portion of a computer program defining the kernel block preferably used in the co-processor of the present invention.

Figure 22 is a portion of a computer program defining the command block preferably used in the co-processor of the present invention.

Figure 23 is a portion of a computer program defining the secret key object preferably used in the co-processor of the present invention.

Figure 24 is a portion of a computer program defining the public keyset preferably used in the co-processor of the present invention.

Figure 25 is a portion of a computer program defining the Diffie-Hellman public keyset preferably used in the co-processor of the present invention.

Figure 26 is a portion of a computer program defining the RSA public keyset preferably used in the co-processor of the present invention.

Figure 27 is a portion of a computer program defining the DSA public keyset preferably used in the co-processor of the present invention.

Figure 28 is a portion of a computer program defining the DSA digital signature preferably used in the co-processor of the present invention.

Figure 29 is a portion of a computer program defining the DSA seed key preferably used in the co-processor of the present invention.

Figure 30 is a portion of a computer program defining the key cache register data type preferably used in the co-processor of the present invention.

Figure 31 is a portion of a computer program defining the symmetrical encryption context store preferably used in the co-processor of the present invention.

Figure 32 is a portion of a computer program defining the one-way hash context store preferably used in the co-processor of the present invention.

Figure 33 is a portion of a computer program defining one example of the CGX wrap code and command interface preferably used in the co-processor of the present invention.

Figure 34 is a portion of a computer program defining the CGX overlay table preferably used in the co-processor of the present invention.

Figure 35 is a portion of a computer program defining the KCS object preferably used in the co-processor of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A block diagram of the cryptographic co-processor hardware formed in accordance with the present invention is illustrated in Figure 1. The cryptographic co-processor is effectively broken down into three major components: Input/Output (I/O) blocks 2, processor blocks 4 and security blocks 6. Preferably, the co-processor may further include a standard direct memory access (DMA) controller circuit 42, which will be described in greater detail. The Input/Output blocks 2 provide several customized functions that enable the application to gain access outside the cryptographic co-processor. The processor blocks 4 make up the central processing unit and control circuitry used to run and control the overall device. The security blocks 6 implement the security features of the cryptographic co-processor as well as protection schemes.

Like most general purpose DSP platforms, the cryptographic co-processor includes components which provide several Input/Output (I/O) functions. These components include a synchronous serial port 12, a bit I/O circuit 8, and an I/O interface circuit 14 to support Personal Computer Memory Card Industry Association (PCMCIA) standard, or Peripheral Component Interconnect (PCI) interfaces. The IDMA I/O interface circuit 14 is a standard 16 bit interface provided directly on the integrated DSP co-processor. The synchronous serial port 12 is a standard serial port which may be used for serial communications and multiprocessor communications. The synchronous serial port 12 may be used to interface to a coder/decoder (CODEC). The I/O pins 8 are used to control external devices such as analog to digital converters and are used for sensing inputs from external devices (e.g., flow control bits, ring detection).

The processor blocks 4 include a DSP 20, reset control circuit 16 and clock control circuit 22. The DSP 20 used in this embodiment is preferably Part No. 2183 manufactured by Analog Devices Inc. of Norwood, MA. However, the secure communication platform may be embedded in any standard DSP or other processor. The 2183 DSP provides the necessary processing power (MIPS) and instructions to implement various applications (e.g., V.34 modem, ADSL, 10 Base T Ethernet). The reset control circuit 16 is a standard circuit used to manage the

reset of the DSP 20 and/or other hardware blocks tied to the cryptographic co-processor platform. The clock control 22 contains a standard system clock which synchronizes data flow. Standard program random access memory (RAM) 10 and data RAM 18 are included in the DSP for storing application programs and data.

The security portion of the co-processor 6 includes an encryption circuit 36, a random number generator circuit 38, a hardware public key accelerator circuit 28, a secure kernel read only memory (ROM) 26, protected kernel random access memory (RAM) 32, volatile key cache registers 34, hash circuit 30 and kernel mode control circuit 24. It is, of course, understood that the encryption circuit 36, random number generator circuit 38, public key accelerator circuit 28, registers 34, hash circuit 30, mode control circuit 24 and other circuits used in the co-processor may be implemented by discrete components or may be equivalently formed as part of the DSP 20, which may be programmed to provide the functions of these circuits. It should also be noted that the term "circuit" used herein incorporates both hardware and software implemented connotations.

The encryption circuit 36 provides a hardware assisted DES engine. A hardware assisted DES engine is provided because it is faster than a software DES engine, which would require more operations to encrypt and decrypt. The DES engine can be used to implement DES and Triple DES encryption and decryption. Furthermore, it implements four cipher modes: Electronic Code Block (ECB), Cipher Block Chaining (CBC), Cipher Feed Back (CFB), and Output Feed Back (OFB). The DES and Triple DES encrypt/decrypt operations are pipelined and preferably execute full 16-round DES in 4 clock cycles. The DES engine preferably encrypts 64 bits of data at a time and has a separate state register 40 (i.e., the feed-back or initialization vector register) that can be read and written. The state register 40 is important in allowing multiple encryption circuit contexts, thus allowing packet switching. With a writable state register 40, a previous context can be reloaded or a new one created. This minimizes the overhead of changing cryptographic keys and initialization vectors. Hardware circuits are provided for padding insertion, verification and removal which accelerates the encryption operation. A control register is provided to program the algorithm and mode to be used.

The hardware random number generator 38 provides a true, non-deterministic noise for the purpose of generating keys, initialization vectors and other random number requirements. Random numbers are preferably 16 bit words provided to the kernel. The secure kernel requests random numbers as needed to perform requested commands and can directly supply from 1 to 65,535 random bytes to a host application.

The hash circuit 30 provides hardware accelerated SHA-1 and MD5 one-way hash processing. The hash circuit is coupled with the encryption circuit 36 where the combined operation chains both hashing and encrypt/decrypt operations to reduce processing time for data which needs both operations applied. For hash-then-encrypt and hash-then-decrypt operations, the cryptographic co-processor performs parallel execution of both functions from the same source and destination buffers. For encrypt-then-hash and decrypt-then-hash operations, the processing is sequential; however, minimum latency is achieved through the pipeline chaining design. An offset can be specified between the start of hashing and the start of encryption to support certain protocols such as IPsec.

The hardware public key accelerator 28 is provided to support large number addition, subtraction, squaring and multiplication. It operates with the secure kernel to provide full public key services to the application program. The kernel provides macro-level algorithms to perform numerous security functions, such as Diffie-Hellman key agreement, Rivest Shamir Adleman (RSA) encrypt or decrypt and calculate/verify digital signatures. The hardware public key accelerator speeds up the computation intensive operations of the algorithms by

providing the mathematical calculations. The secure kernel is embodied as firmware which is mask programmed into a ROM. There are preferably 32K words of kernel ROM 26 available for a cryptographic library and an Application Programming Interface (API).

A kernel mode control circuit 24 is provided for controlling a security perimeter around the cryptographic hardware and software. Because the cryptographic co-processor has a general purpose DSP and a cryptographic co-processor, the device may operate in either a user mode or a kernel mode. In the user mode the kernel space is not accessible, while in the kernel mode it is accessible. When in the kernel mode, the kernel RAM and certain registers and functions are accessible only to the secure kernel firmware. The kernel executes host requested macro level functions and then returns control to the calling application.

The protected kernel RAM 32 provides a secure storage area on the cryptographic co-processor for sensitive data such as keys or intermediate calculations during public key operations. The kernel mode control circuit 24 controls access by only allowing the internal secure kernel mode access to this RAM. A public keyset and a cache of 15 secret keys may be stored in the protected kernel RAM 32. The purpose of having a separate area for volatile key RAM is security related. It isolates the RAM from the application thus making it more difficult to accidentally leak RED (plaintext) key material.

A key feature of the co-processor of the present invention is its universality. First, a manufacturer may substitute the co-processor of the present invention for a conventional digital signal processor (DSP) in the equipment (e.g., modem, cellular phone, etc.) which is being manufactured. The conventional digital signal processor (DSP) does not provide secure communications. However, the co-processor of the present invention performs all of the functions of the conventional DSP but also has the unique capability of providing secure communications.

Secondly, the co-processor of the present invention is also universal in that it provides a library of cryptographic algorithms which may be selected by the manufacturer for use in the digital communications equipment that is being manufactured. The manufacturer may select one or more encryption algorithms or

functions (e.g., DES, HASH functions, etc.) for particular use with the equipment being manufactured. The manufacturer uses one or more commands to access the library and select the desired encryption algorithm or function from the library. The advantage of the universal co-processor is that the user does not have to recreate any encryption or hashing or public key algorithms, as they are pre-programmed in the ROM library; all the manufacturer has to do is select whichever algorithm he wishes to use in the encryption or hashing process. Selection of a particular algorithm is facilitated by using a pre-programmed command set which includes a number of encryption, public key and HASH commands. For example, the command CGX-PUBKEY-ENCRYPT refers to a public key encrypt command which is used to encrypt the application's data using an RSA encryption algorithm which is stored in the library. This operation implements encryption or the RSA signature operation using the pubkey member of a public key structure.

These commands are recognized by a microprocessor forming part of the co-processor, which accesses the library and retrieves the particular encryption algorithm, for example, the RSA algorithm, from the library and uses it in the encryption process.

The application software designer selects the encryption and/or HASH functions from the library and thus avoids having to directly communicate with the crypto hardware.

Because the algorithms are mask programmed into the read only memory (ROM) and are on the same platform, i.e., substrate, as the DSP, they are more secure than if they were embodied in pure software.

As mentioned previously, a standard direct memory address (DMA) controller circuit 42 is preferably included within the cryptographic co-processor to facilitate bulk data movements without requiring continuous processor supervision. It preferably allows 32 bit transfers to occur at up to 40 M words per second. The DMA controller circuit 42 is coupled to the input/output section 2 and the security section 6 of the co-processor.

An interrupt controller circuit 50 is coupled to the security blocks 6 and the processor blocks 4. The interrupt controller circuit 50 provides enhancements to the existing interrupt functions in the processor blocks 4. The interrupt controller

circuit 50 provides interrupt generation capabilities to the processor blocks 4 or to a standard external host processor. Under programmable configuration control, an interrupt may be generated due to the completion of certain operations such as encryption complete or hash complete.

An application register circuit 52 is coupled to the security blocks 6 and the processor blocks 4. The application register circuit 52 is a set of memory-mapped registers which facilitate communications between the processor blocks 4 and a standard host processor via the PCI or PCMCIA bus. One of the registers is preferably 44 bytes long and is set as a 'mailbox' up to hold a command structure passed between standard host processor and the processor blocks 4. The application register circuit 52 also provides the mechanism which allows the processor blocks 4 and the standard host processor to negotiate which has ownership of the hash block circuit 30 and the encryption block circuit 36.

An external memory interface circuit 54 is coupled to the security blocks 6, the processors blocks 4 and the direct memory address (DMA) controller circuit 42. The external memory interface circuit 54, provides an interface to external memory. The preferred address width is 26 bits wide and the preferred data width is 32 bits wide.

A laser variable storage circuit 56, is coupled to the security blocks 6 and the processor blocks 4. The laser variable storage circuit 56 consists of 256 bits of tamper-proof factory programmed data which is preferably accessible to the processor blocks 4 and the secure kernel. Included in the laser variable bits are: 112-bit local storage variable (master key encryption key), 80-bit randomizer seed, 48-bit program control data (enables/disables various IC features and configures the IC), and 16-bit standard cyclic redundancy check (CRC) of the laser data. The program control data bits (PCDB) preferably includes configuration for permitted key lengths, algorithm enables, red key encryption key loading and internal IC pulse timing characteristics. Some of the PCDB settings may be overridden with a digitally signed token which may be loaded into the cryptographic co-processor when it boots. These tokens are created by the IC manufacturer and each is targeted to a specific IC, using a hash of its unique identity (derived from the above laser variable).

A serial electrically erasable programmable read-only memory (EEPROM) interface circuit 58 is coupled to the security blocks 6 and the processor blocks 4. The serial EEPROM interface circuit 58 is used to allow an external non-volatile memory to be connected to the cryptographic co-processor for the purpose of storing PCI or PCMCIA configuration information (plug and play), as well as general purpose non-volatile storage. For example, encrypted (black) keys could be stored into EEPROM for fast recovery after a power outage.

The cryptographic co-processor may be integrated into a wide variety of systems, including those which already have a processor and those which will use the cryptographic co-processor as the main processor. The cryptographic co-processor and more specifically the input/output blocks 2, can be configured to one of three host bus modes: IDMA 72, PCI 76 or PCMCIA 76. All three of these interface modes are industry standards. A bus mode input 66 and a bus select input 68 is coupled to the input/output blocks 2 for selecting the mode.

The bus mode input 66 is also coupled to a multiplexing circuit 60. The multiplexing circuit 60 is a standard multiplexing circuit used to select between one of the three host bus modes (IDMA, PCI, PCMCIA).

An external memory interface (EMI) 70 is a standard memory interface used to interface with external devices. This interface is coupled to the processor block 4.

An interrupt input circuit 62 is coupled to the processor blocks 4. The interrupt input circuit 62 is a standard interrupt input circuit provided for the use by external devices. A standard flag input/output circuit 64 is also coupled to the processor blocks 4.

The master key (LSV) is intended to be different on each chip. Therefore, it is not mask-programmed. Mask programming is done with lithography which affects the artwork of the integrated circuit. Since no two chips should have the same LSV, the LSV is preferably programmed into a memory of the chip by laser trimming. Laser trimming is advantageous in that each die may be fabricated with a different LSV so that each cryptographic co-processor is individualized with a particular LSV.

The following is a detailed description of the cryptographic co-processor taken from a User's Manual prepared by the assignee and owner of the invention, Information Resource Engineering, Inc. (IRE). The cryptographic co-processor is often referred to herein by the trademark '*CryptIC*'. The co-processor is also often referred to by part number ADSP 2141. The following includes a description of each of the major subsystems within the CryptIC co-processor, including complete register details.

GENERAL DESCRIPTION

It can be seen from Figure 1A that the ADSP 2141 *CryptIC* is a highly integrated Security Processor ASIC which incorporates a sophisticated, general purpose DSP, along with a number of high-performance Cryptographic function blocks, as well as a PCI, PCMCIA and Serial EEPROM interface. It is fabricated in .35 μ CMOS triple-layer metal technology utilizing a 3.3V Power Supply. It is initially available in a 208-pin MQFP package with a Commercial (0° – 70°C) Temperature Range. A 208-pin TQFP package will follow.

DSP Core

The DSP Core is a standard Analog Devices ADSP-2183 with full ADSP-2100 family compatibility. The ADSP-2183 combines the base DSP components from the ADSP-2100 family with the addition of two serial ports, a 16-bit Internal DMA port, a Byte DMA port, a programmable timer, Flag I/O, extensive interrupt capabilities, and on-chip program and data memory. The External Memory Interface of the 2183 has been extended to support up to 64M-words addressing for both Program and Data memory. Some core enhancements have been added in the *CryptIC* version, including on-chip Security ROM and Interrupt functions.

Refer to the Analog Devices ADSP-2183 Rev B datasheet for further information. (Available in Adobe Acrobat format at:
http://www.analog.com/pdf/ADSP_2183.pdf)

Secure Kernel (Firmware)

The Secure Kernel is embodied as firmware which is mask-programmed into ROM within the DSP, thus rendering it tamper-proof. The Kernel provides the API (Application Programming Interface) to applications which require security services from the *CryptIC*. Those applications may be software executing in the 'User Mode' on the DSP, or they may be external 'Host' software accessing the *CryptIC* via a PCI or PCMCIA bus. Approximately 40 Crypto commands – called CGX (CryptoGraphic eXtensions) – are provided at the API and a simple Control Block structure is used to pass arguments into the secure Kernel and return Status.

The Secure Kernel firmware runs under a 'Protected Mode' state of the DSP as described below in section 0. This guarantees the security integrity of the system during the execution of Kernel processes and, for example, prevents disclosure of Cryptographic Key data or tampering with a security operation.

Kernel Mode Control

The Kernel Mode Control block is responsible for enforcing the 'Security Perimeter' around the cryptographic functions of the *CryptIC*. The device may either be operating in 'User Mode' (Kernel Space is not accessible) or 'Kernel Mode' (Kernel Space is accessible) at a given time. When in the Kernel mode, the Kernel RAM and certain protected Crypto registers and functions (Kernel Space) are accessible only to the Secure Kernel firmware. The Kernel executes Host-requested Macro-level functions and then returns control to the calling application. The Kernel Mode Control hardware subsystem will reset the DSP should any security violation occur, such as attempting to access a protected memory location while in User mode. (A readable register reports the memory address of the violation for debug purposes.)

Protected Kernel RAM

The 4K x 16 Kernel RAM provides a secure storage area on the *CryptIC* for sensitive data such as Keys or intermediate calculations during Public Key operations.

The Kernel Mode Control block (above) enforces the protection by only allowing internal Secure Kernel Mode accesses to this RAM. A Public Keyset and a cache of up to 15 Secret keys may be stored in Kernel RAM. Secure Key storage may be expanded to 700 Secret Keys by assigning segments of the ADSP2183 internal Data RAM to be 'Protected'. This is accomplished via a *CGX_INIT* command argument.

Encrypt Block

The Encrypt Block performs high-speed DES and Triple DES encrypt/decrypt operations. All 4 standard modes of DES are supported: Electronic Code Book (ECB), Cipher Block Chaining (CBC), 64-bit Output Feedback (OFB) and 1-bit, 8-bit and 64-bit Cipher Feedback (CFB). The DES encrypt/decrypt operations are highly pipelined and execute full 16-round DES in only 4 clock cycles. Hardware support for Padding insertion, verification and removal further accelerates the encryption operation. *Context Switching* is provided to minimize the overhead of changing crypto Keys and IV's to nearly zero.

Hash Block

The Secure Hash Block is tightly coupled with the Encrypt Block and provides hardware accelerated one-way Hash functions. Both the MD-5 and SHA-1 algorithms are supported. Combined operations which chain both Hashing and Encrypt/Decrypt functions are provided in order to significantly reduce the processing time for data which needs both operations applied. For Hash-then-Encrypt and Hash-then-Decrypt operations, the *CryptIC* can perform parallel execution of both functions from the same source and destination buffers. For Encrypt-then-Hash and Decrypt-then-Hash operations, the processing must be sequential, however minimum latency is still provided through the pipeline chaining design. An Offset may be specified between the start of Hashing and the start of Encryption to support certain protocols such as IPsec, and 'Mutable bit' handling is provided in hardware.

Random Number Generator (RNG) Block

The hardware Random Number Generator provides a true, non-deterministic noise source for the purpose of Generating Keys, Initialization Vectors (IV's), and other random number requirements. Random numbers are provided as 16-bit words to the Kernel. The Security Kernel requests Random Numbers as needed to perform requested CGX commands such as *CGX_Gen Key*, and can also directly supply from 1 to 65,535 Random Bytes to a host application via the *CGX_Random* CGX command.

Public Key Accelerator

The Public Key Accelerator module works in concert with the CGX Secure Kernel firmware to provide full Public Key services to the host application. The CGX Kernel provides Macro-level library functions to perform Diffie-Hellman Key Agreement, RSA Encrypt or Decrypt, Calculate and Verify Digital Signatures, etc. The hardware accelerator block speeds the computation-intensive operations such as 32 x 32 multiply, 32-bit adds/subtracts, Squaring, etc..

PCMCIA/□Processor Interface

A standard 16-bit PCMCIA interface is provided directly on the *CryptIC*. This interface may also be used in certain applications as a generic 16-bit □Processor Interface.

PCI/Cardbus Interface

A full 66/33MHz PCI v2.1 bus interface has been added to the core DSP functions. The 32-bit PCI interface supports both Bus Master and Target modes. The *CryptIC* is capable of using DMA to directly access data on other PCI entities and pass that data through its Encryption/Hash engines.

32-Bit DMA Controller

The *CryptIC* incorporates a high-performance 32-bit DMA controller which can be set-up to efficiently move data between Host PCI memory, the Hash/Encrypt blocks, and/or External Memory. The DMA controller can be used with the PCI bus in Master mode, thus autonomously moving 32-bit data with minimal DSP intervention. Up to 255 long words (1020 bytes) can be moved at a time.

Application Registers

The Application Registers are a set of memory-mapped registers which facilitate communications between the *CryptIC* DSP and a Host processor via the PCI or PCMCIA bus. One of the Registers is 44 bytes long and is set-up to hold the CGX command structure passed between the Host and DSP processors. The Application Registers also provide the mechanism which allows the DSP to arbitrate whether it or the DMA controller (Host) has ownership of the External Memory interface.

Serial EEPROM Interface

The Serial EEPROM interface is used to allow an external non-volatile memory to be connected to the *CryptIC* for the purpose of storing PCI or PCMCIA configuration information (Plug and Play), as well as general-purpose non-volatile storage. For example, encrypted (Black) Keys or a digital certificate could be stored into EEPROM for fast recovery after a power outage.

Interrupt Controller

The Security Block Interrupt Controller provides enhancements to the existing Interrupt Functions in the ADSP 2183 core. Primarily, the Interrupt Controller provides a new Interrupt Generation capability to the DSP or to an external Host Processor. Under programmable configuration

control, a 'Crypto Interrupt' may be generated due to completion of certain operations such as *Encrypt Complete*, *Hash Complete*, etc. The interrupt may be directed either at the DSP core, or provided on an output line (PF7) to a Host subsystem.

5 **Laser Variable Storage**

The Laser Variable Storage consists of 256 bits of Tamper-Proof Factory programmed data which is only accessible to the internal function blocks and the Security Kernel. Included in these Laser Variable bits are:

- 112-bit Local Storage Variable (Master Key-Encryption-Key)
- 10 • 80-bit Randomizer Seed
- 48-bits Program Control Data (Enables/Disables various IC features and configures the IC)
- 16-bit CRC of the Laser Data

15 The Program Control Data (PCD) bits include configuration for permitted Key Lengths, Algorithm Enables, Red KEK loading, Internal IC Pulse Shaping Characteristics, etc. Some of the PCD settings may be overridden with a Digitally Signed Token which may be loaded into the *CryptIC* when it boots. These Tokens are created by IRE and each is targeted to a specific *CryptIC* using a Hash of its unique identity (derived from the above Laser Variable).

20

Downloadable Secure Code

The *CryptIC* is designed to allow additional Security Functions to be added to the device through a *Secure Download* feature. Up to 16k words of code may be downloaded into internal memory within the DSP and this code can be given the security privileges of the Kernel firmware. All downloaded firmware is authenticated with a Digital Signature and verified with an on-chip Public Key. Additional functions could include new Encryption, Hash or Public Key algorithms such as IDEA, RC-4, RIPEMD, Elliptic Curve, etc.

25

MEMORY CONFIGURATION

The *CryptIC* provides a large amount of on-chip 0 wait-state RAM, a block of mask-programmed ROM and also provides an external memory bus interface in order to allow a considerable expansion using off-chip devices. The on-chip RAM consists of three separate groups: 16k x 24 of Internal Program RAM, 16k x 16 of Internal Data RAM, and 4k x 16 of Kernel RAM.

Memory Map

The *CryptIC* memory map is very similar to that of the ADSP 2183, except that it includes significantly more Off-Chip memory addressing, and has additional Crypto Registers which are accessible to the User. The *CryptIC* memory maps are shown in Figures 2-4 of the drawings.

The **PMOVLAY** register is responsible for selecting 8k-word 'Pages' of upper Program Memory, as shown in the table below.

PMOVLAY register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	X	x	x	x	x	x	x	x	X	x	x	x	x	X
msb											Lsb				
PMOVLAYH External Address											PMOVLAYL Page Select				

The 4 lsb's (bits 3:0) are interpreted as follows:

1111	F	Kernel ROM 0 (Base Page)
1110	E	Kernel ROM 1
1101	D	Kernel ROM 2
1100	C	Kernel ROM 3 (Top)
1011	B	reserved
.	.	.
0011	3	reserved
0010	2	External RAM Odd Pages
0001	1	External RAM Even Pages
0000	0	Internal RAM

The 12 msb's (bits 15:4) are mapped to the most-significant external address pins on the *CryptIC* (addr 25:14).

Thus, to address Kernel ROM page 1, the PMOVLAY register should be set to 0x000E (although the uppermost 12 bits are ignored in this case). To address External memory page 38, the PMOVLAY register should be set to 0x0131 (0x013 are the 12 msb's representing pages 38 & 39 and a 1 in the least-significant nibble indicates External even page).

The **DMOVLAY** register is responsible for selecting 8k-word 'Pages' of lower Data Memory, as shown in the table below.

DMOVLAY register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	X	x	x	x	x	x	x	x	X	x	x	x	x	x
msb											lsb				
DMOVLAYH External Address											DMOVLAYL Page Select				

The 4 lsb's (bits 3:0) are interpreted as follows:

1111	Kernel RAM and Kernel Registers
1110	reserved
1101	reserved
.	.
.	.
.	.
0011	reserved
0010	External RAM Odd Pages
0001	External RAM Even Pages
0000	Internal RAM

The 12 msb's (bits 15:4) are mapped to the most-significant external address pins on the *CryptIC* (addr 25:14).

Thus, to address the Kernel RAM/Crypto Registers page, the DMOVLAY register should be set to 0x000F (although the uppermost 12 bits are ignored in this case). To address External memory page 159 (decimal), the DMOVLAY register should be set to 0x04F2 (0x04F are the 12 msb's representing pages 158 & 159 and a 2 in the least-significant nibble indicates External odd page).

Register Set

The *CryptIC* contains a number of additional registers (beyond those in a ADSP 2183) which are mapped into the ADSP 2183's external data memory space. Some of the Registers are intended to be accessed only by the Secure Kernel and are referred to as **Protected Registers** (Table 2 below). (In some cases, designers may require features of the *CryptIC* which are only available in Protected Registers.) The Registers which are accessible either to the DSP running in the User mode or to an outside PCI/PCMCIA Bus entity are referred to as **Unprotected Registers** and are listed Table 1 below.

All of the Protected Registers are memory-mapped in the Data Memory space of the DSP at 0x1000 – 0x17FF. The Unprotected registers reside at 0x1800 – 0x1FFF. The **DMOVLAY** register must be set to 0x000F to access these registers. From the Host perspective, the base register address in the PCI/PCMCIA space is set by the BASEADDR register.

Note that although the DSP cannot directly read 32-bit registers, it can perform a 32-bit DMA operation from a 32-bit register into its own external memory space. See section on 32-bit DMA Controller, described under a main heading below.

In the table below, 16-bit address refers to the DSP and 32-bit address refers to PCI host.

UNPROTECTED REGISTERS

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset Default	DESCRIPTION
APPLICATION REGISTERS					
0x1880-0x1895	0x0000-2B	CGX Command	R/W		44-byte CGX command register
0x18A0	0x0040-41	Status	R	0x0000	Application status
0x18A1	0x0042-43	Lock	R/W		DSP/Host lock control
0x18A2	0x0044-47	Misc Status	R/W		Miscellaneous status bits: DSP <--> host
0x18A3	N/A	Select Delay	R/W		Delay configuration for memory pulse generation
0x18A4	N/A	Hash/Encrypt Byte Enable	R/W		Byte enables for data R/W to Hash/Encrypt block
0x18A5	N/A	Reset Violation	R	0x0000	Holds the memory type and Address of

		Memtype/Addr			the last protection violation-induced Reset
0x18A6	0x004C-4F	Extmem Config	R/W		External memory configuration
DMA & PCI REGISTERS					
DSP-Visible Registers:					
0x1840	N/A	Host Address [15:0]	R/W		Lower 16-bits of Host Address
0x1841	N/A	Host Address [31:16]	R/W		Upper 16-bits of Host Address
0x1842	N/A	Local Address [15:0]	R/W		Lower 16-bits of Local Address
0x1843	N/A	Local Address [31:16]	R/W		Upper 16-bits of Local Address
0x1844	N/A	Command	R/W		Command
0x1845	N/A	Status/Config.	R/W		Status/Configuration Register
0x1846	N/A	PCI Core Status/Config.	R/W		Status/Configuration for PCI core
0x1847	N/A	PCI Extmem Status	R		PCI External Memory Status
PCI Host-Visible Registers:					
N/A	0x00C0-C3	PCI Target Page	R/W		Target Page specifier when <i>CryptIC</i> External Memory is Target
N/A	0x00C4-C7	PCI Target Read Count	R/W		Maximum number of dwords for Target Read Transfer
N/A	0x00C8-CB	Endian	R/W		Big/Little Endian select
PCMCIA Host-Visible Registers:					
N/A	0x00C0-C1	DMA Address	R/W		DMA address register
N/A	0x00C4-C5	Start/Stop DMA	R/W		DMA control
N/A	0x00C8-CB	Endian	R/W		Big/Little Endian select

UNPROTECTED REGISTERS (cont.)

HASH/ENCRYPT REGISTERS					
Configuration Registers:					
Protected	0x0200	H/E control	R/W	0x0000	Hash/Encrypt block control word
Protected	0x021E	Pad control	R/W	0x0000	Pad control word
Protected	0x0220-0223	Length	R/W	0x0000	32-bit data length, in bytes
Protected	0x0224	Offset	R/W	0x0000	Offset, (0 to 15 in dwords), from start of hash (encryption) to start of encryption (hash)
Protected	0x0226	Control	R/W	0x0007	Operation control.
Protected	0x0228	Consume Pad	W		Command to consume final pad block
Status Registers:					
0x1016	0x022C	Pad Status 0	R		Decrypted next header byte, # Pad bytes, Context 0
0x1017	0x022E	Pad Status 1	R		Decrypted next header byte, # Pad bytes,

					Context 1
Protected	0x0230	GeneralStatus	R		Status result from Hash/Encrypt operation
Protected	0x0232	ControlReady	R		1 = input ready for new control word, 0 = not ready
Protected	0x0234	DataReady	R		1 = input ready for data FIFO, 0 = not ready
Protected	0x0236	StatFreeBytes	R		Number free input bytes in crypto FIFO (in 64-bit blocks)
Protected	0x0238	StatOutBytes	R		Number output bytes ready in crypto FIFO (in 64-bit blocks)
Context 0 Registers:					
Protected	0x0240-0247	Key3_0	W		Key 3, for Triple DES: Crypto Context 0
Protected	0x0248-024F	Key2_0	W		Key 2, for Triple DES: Crypto Context 0
Protected	0x0250-0257	Key1_0	W		Key 1, for Triple DES or DES: Crypto Context 0
Protected	0x0258-025F	Salt_0	W		IV for key decryption: Crypto Context 0
Protected	0x0260-0267	IV_0	R/W		IV for data encrypt/decrypt: Crypto Context 0
Protected	0x0268-027B	Digest_0	R/W		(Inner) Digest: Crypto Context 0
Protected	0x027C-028F	OuterDigest_0	W		Outer Digest: Crypto Context 0
Protected	0x0290-0293	HashByteCnt_0	R/W		Starting byte count, for hash resume: Crypto Context 0
Context 1 Registers:					
Protected	0x02A0-02A7	Key3_1	W		Key 3, for Triple DES: Crypto Context 1
Protected	0x02A8-02AF	Key2_1	W		Key 2, for Triple DES: Crypto Context 1
Protected	0x02B0-02B7	Key1_1	W		Key 1, for Triple DES or DES: Crypto Context 1
Protected	0x02B8-02BF	Salt_1	W		IV for key decryption: Crypto Context 1
Protected	0x02C0-02C7	IV_1	R/W		IV for data encrypt/decrypt: Crypto Context 1
Protected	0x02C8-02DB	Digest_1	R/W		(Inner) Digest: Crypto Context 1
Protected	0x02D0-02EF	OuterDigest_1	W		Outer Digest: Crypto Context 1
Protected	0x02F0-02F3	HashByteCnt_1	R/W		Starting byte count, for hash resume: Crypto Context 1
Data In/Out FIFOs:					
Protected	0x0380	Data FIFO	R/W		FIFO: Data In/Data Out
INTERRUPT CONTROLLER REGISTERS					
DSP-Visible Registers:					
0x1800	N/A	DSP Unmasked Status	R		Interrupt source current states – Prior to mask
0x1801	N/A	DSP Masked Status	R		Interrupt source current states – Post mask

0x1801	N/A	DSP Clear Int	W		Clear selected Interrupt
0x1802	N/A	DSP Mask Control	R/W		Interrupt mask register
0x1803	N/A	DSP Int Config.	R/W		DSP Interrupt configuration register
0x1804	N/A	Force Host Int	W		Force interrupt to Host (PCI/PCMCIA)
0x1805	N/A	H/E Error Code	R		Provides the H/E Error Code

UNPROTECTED REGISTERS (cont.)

Host-Visible Registers:					
N/A	0x0080-0081	Host Unmasked Status	R		Interrupt source current states – Prior to mask
N/A	0x0084-0085	Host Masked Statu	R		Interrupt source current states – Post mask
N/A	0x0084-0085	Host Clear Int	W		Clear selected interrupt
N/A	0x0088-0089	Host Mask Control	R/W		Interrupt mask register
N/A	0x008C-008D	Host Int Config.	R/W		Host interrupt configuration register
N/A	0x0090-0091	Force DSP Int	W		Force interrupt to DSP
N/A	0x0094-0095	H/E Error Code	R		Provides the H/E Error Code
IDMA INTERFACE REGISTERS					
Host-Visible Registers:					
N/A	0x00A0-A1	IDMA Indirect Address	W		Address latch for IDMA Indirect transfers
N/A	0x00A4-A5	IDMA Config	R/W		IDMA Configuration (Direct or Indirect)
N/A	0x8000-FFFF	IDMA Data	R/W		32K IDMA Data Range
SERIAL EEPROM REGISTERS					
0x1900	N/A	Device ID	R/W		16-bit PCI device ID
0x1901	N/A	Vendor ID	R/W		16-bit PCI vendor ID (11D4h)
0x1902	N/A	Rev ID/Class	R/W		8-bit chip Revision ID, 8-bits of PCI Class Code
0x1903	N/A	Class Code	R/W		remaining 16-bits of PCI Class Code
0x1904	N/A	Header Type/Int	R/W		PCI header type & Interrupt Pin
0x1905	N/A	Subsystem ID	R/W		16-bit PCI Subsystem ID
0x1906	N/A	Subsystem Vendor ID	R/W		16-bit Subsystem Vendor ID
0x1907	N/A	Max Lat, Min Gnt	R/W		Maximum Latency, Min Grant parameters
0x1908	N/A	Cardbus1	R/W		lower 16-bits of Cardbus CIS pointer
0x1909	N/A	Cardbus2	R/W		upper 16-bits of Cardbus CIS pointer
0x190A	N/A	Baddr mask1	R/W		Specifies 1 = modifiable 0 = our addresses
0x190B	N/A	Baddr mask2	R/W		Upper 16 bits
0x190C	N/A	CIS Size	R/W		CIS Size spec 16-bit (Upper 8 bits are 0)
0x190F	N/A	Cmd/Status	R/W		EEPROM Command and Status Register

Table 1 *CryptIC Unprotected Register Set*

PROTECTED REGISTERS

ADDRESS (16 BIT)	REGISTER NAME	R/W	Reset Default	DESCRIPTION
HASH/ENCRYPT REGISTERS				
Configuration Registers:				
0x1000	H/E control	R/W	0x0000	Hash/Encrypt block control word
0x100F	Pad control	R/W	0x0000	Pad control word
0x1010-1011	Length	R/W	0x0000	32-bit data length, in bytes
0x1012	Offset	R/W	0x0000	Offset, (0 to 15 in dwords), from start of hash (encryption) to start of encryption (hash)
0x1013	Control	R/W	0x0007	Operation control.
0x1014	Consume Pad	W		Command to consume final pad block
Status Registers:				
0x1016	Pad Status 0	R		Decrypted next header byte, # Pad bytes, Context 0
0x1017	Pad Status 1	R		Decrypted next header byte, # Pad bytes, Context 1
0x1018	GeneralStatus	R		Status result from Hash/Encrypt operation
0x1019	ControlReady	R		1 = input ready for new control word, 0 = not ready
0x101A	DataReady	R		1 = input ready for data FIFO, 0 = not ready
0x101B	StatFreeBytes	R		Number free input bytes in crypto FIFO (in 64-bit blocks)
0x101C	StatOutBytes	R		Number output bytes ready in crypto FIFO (in 64-bit blocks)
Context 0 Registers:				
0x1020-1023	Key3_0	W		Key 3, for Triple DES: Crypto Context 0
0x1024-1027	Key2_0	W		Key 2, for Triple DES: Crypto Context 0
0x1028-102B	Key1_0	W		Key 1, for Triple DES or DES: Crypto Context 0
0x102C-102F	Salt_0	W		IV for key decryption: Crypto Context 0
0x1030-1033	IV_0	R/W		IV for data encrypt/decrypt: Crypto Context 0
0x1034-103D	Digest_0	R/W		(Inner) Digest: Crypto Context 0
0x103E-1047	OuterDigest_0	W		Outer Digest: Crypto Context 0
0x1048-1049	HashByteCnt_0	R/W		Starting byte count, for hash resume: Crypto Context 0
Context 1 Registers:				
0x1050-1053	Key3_1	W		Key 3, for Triple DES: Crypto Context 1
0x1054-1057	Key2_1	W		Key 2, for Triple DES: Crypto Context 1
0x1058-105B	Key1_1	W		Key 1, for Triple DES or DES: Crypto Context 1
0x105C-105F	Salt_1	W		IV for key decryption: Crypto Context 1
0x1060-1063	IV_1	R/W		IV for data encrypt/decrypt: Crypto Context 1
0x1064-106D	Digest_1	R/W		(Inner) Digest: Crypto Context 1
0x106E-1077	OuterDigest_1	W		Outer Digest: Crypto Context 1
0x1078-1079	HashByteCnt_1	R/W		Starting byte count, for hash resume: Crypto Context 1
Data In/Out FIFOs:				
0x10C0	Data FIFO	R/W		FIFO: Data In/Data Out
KEY MANAGEMENT REGISTERS				
0x1180	RAM control	R/W		Selects the current KRAM owner: DSP, PK, DMA
0x1181	PM Reserve	R/W		Selects 1kword segments of PM to 'Protect' into

				Kernel
0x1182	DM Reserve	R/W		Selects 1kword segments of DM to 'Protect' into Kernel
0x1183	KM RAM Reserve	R/W		Selects 256-byte segments of KM to 'Protect' into Kernel
0x1184	Hash/Enc Control	R/W		Selects owner of H/E block: DSP or Host
0x1185	Reset Control	R/W		Allows internal reset of H/E, PK, RNG

Table 2 *CryptIC Protected Register Set***BUS INTERFACES**

The *CryptIC* supports multiple bus interfaces in order to allow it to be integrated into a wide variety of host systems. These buses are:

- 5 • Host Processor bus
 - ⇒ PCI (also Cardbus) - or -
 - ⇒ PCMCIA - or -
 - ⇒ 2183 IDMA
- External Memory Interface (EMI) bus

10 These buses will be described in the following sections.

Host Bus Mode Selection

The *CryptIC* Host Bus may be configured for one of 4 personalities: ADSP 2183 Compatible IDMA Mode, IDMA Enhanced Mode, PCI Bus Mode, or PCMCIA Bus Mode. The selection of mode is made with 2 Hardware control inputs

15 **BUS_MODE** and **BUS_SEL** at boot time.

Bus Mode <input type="checkbox"/> Pins <input type="checkbox"/>	BUS_MODE	BUS_SEL
2183 IDMA Compatible Mode	0	0
2183 IDMA Enhanced Mode	0	1
PCI Bus Mode	1	0
PCMCIA Bus Mode	1	1

Table 3 *Bus Mode Selection*

A number of pins on the *CryptIC* are internally multiplexed in order to change bus personalities. Refer to the *CryptIC* Datasheet for details.

20

This selection may not be changed after the *CryptIC* comes out of power-up Reset. It is typically expected that the Bus Mode signals are tied to ground or V_{DD} on the PC Board.

5

PCI/Cardbus Host Processor Bus

When the *CryptIC* is configured for the PCI host bus mode, the Multiplex bus pins become personalized to directly connect to a 3.3V PCI local bus. The PCI core on the *CryptIC* is compliant with version 2.1 of the standard and supports a 32-bit wide bus. The PCI clock speed may be run from 10MHz to 66MHz.

10

PCI Interface Specifications

The *CryptIC*'s PCI 'core' meets the following specifications:

- PCI Version 2.1
- Target / Master transfer capability
- Configuration Space Read/Write
- 15 • Memory Mode Read/Write – Single word or Burst transfer
- Abort and auto re-try

The PCI interface does NOT support the following:

- I/O Mode Read/Write
- Fast Back-to-Back transactions
- 20 • Memory Write Invalidate operations

PCI Address Map

As shown in Figure 5, the *CryptIC* appears on the PCI Bus as a single contiguous memory space of 128k bytes.

The *CryptIC* presents a 17 bit [16:0] address interface as a PCI Target.

25

Inbound PCI address bits [31:17] are decoded by the *CryptIC* PCI core to determine

whether or not the PCI access matches the PCI Memory *Base Address Register*, thus determining whether the access is to the *CryptIC* or not. However, bits [31:17] are not reflected in the *CryptIC* register addresses.

Once the *CryptIC*'s PCI address has been decoded, the next most significant address bit, [A16], determines whether the lower 16 address bits should be decoded as an internal *CryptIC* register/memory address or reflected to the *CryptIC*'s external memory interface. If address bit 16 is 0, the 16 lsb's are interpreted as *CryptIC* internal register/memory address bits. If address bit 16 is 1, the 16 lsb's [A15-A0] are combined with the 11-bit page designator in the PCI *Target Page Register* to form the external memory address. The PCI Target Page register is addressable by the Host through the PCI Interface and specifies the 11 upper address bits for PCI transfers from/to external memory. See section on PCI Target Page Register (TARGADCNT) described further below.

PCI Target Mode Transfers

As a PCI Target entity, the *CryptIC* provides memory-mapped or I/O-mapped access to its 'unprotected' memory and register space. This includes read/write access through the *CryptIC* to the external memory connected to the EMI bus.

For all Target mode transfers, the *CryptIC* DMA engine is called upon to perform the data movements inside the *CryptIC* between the PCI core and the desired memory or register location(s). This DMA action is automatic and the initiating PCI entity is unaware of the DMA participation in the transfer. It is important however to note the DMA's target transfer role as it effects other DSP-initiated DMA operations. Since Target transfers initiated from other PCI entities are typically unaware of other DMA activities occurring within the *CryptIC*, the DMA arbiter gives precedence to Target DMA transfers. A DMA transfer in-process will not be preempted, however any pending DSP-initiated DMA will be deferred until after all Target transfers have been completed. (A status register in the DMA controller allow the DSP to determine

whether it has seized the controller or whether a Target transfer is running.) Refer also to 0 for more information on the DMA controller.

In addition, in order for Target transfers to occur to/from External Memory, the DSP must grant ownership of the External Memory bus to the DMA engine. If the
 5 Ext. mem. bus is not granted, then the data written to External memory will be lost and a read will return invalid data. See section 0 for more information.

PCI Target transfers to/from the *CryptIC* may in some cases experience a timeout abort and re-start due to latencies in the PCI core FIFO's, address/data setup times, memory wait-states, etc.. These are more likely to occur with reads than writes
 10 due to the 'round-trip' nature of a read. In fact, if writes are kept to 8 dwords or fewer, then timeout aborts can be avoided, since the PCI core write FIFO (12 dwords) can store the written data until it can be DMA'ed to its destination within the *CryptIC*.

The interaction of the clock speeds of the PCI bus and of the *CryptIC* core must also be considered. Ideally, the *CryptIC* core clock should be equal to or faster
 15 than the PCI bus clock in order to allow it to unload incoming PCI data at least as fast as it arrives. If this is the case, then only IDMA transfers or transfers to external memory with >1 wait state will result in PCI timeout aborts for transfers >8 dwords.

PCI Master Mode Transfers

The *CryptIC* can use PCI Master Mode transfers for the most efficient transfer
 20 of data into or out of the device. Master mode transfers are always performed under control of the DSP. Refer also to 0 for more information on the DMA controller.

PCI Core Configuration Registers

As viewed from the PCI Host perspective, the 256-byte PCI configuration space is defined below in Table 4. The fields marked xxxxxxxx are 'don't cares'.
 25 Shaded fields are read-only registers and are loaded from the serial EEPROM connected to the *CryptIC*.

31	16			15	0	Addr.
Device ID				Vendor ID		00h
Status				Command		04h
Class Code				Revision ID		08h
BIST		Header Type		Master Latency Timer	Cacheline Size	0Ch
I/O Base Address						10h
Memory Base Address						14h
Reserved (Dual Base)						18h
Reserved						1Ch
XXXXXXXX		XXXXXXXX		xxxxxxxx	xxxxxxxx	20h
XXXXXXXX		XXXXXXXX		xxxxxxxx	xxxxxxxx	24h
XXXXXXXX		XXXXXXXX		xxxxxxxx	xxxxxxxx	28h
Subsystem ID				Subsystem Vendor ID		2Ch
XXXXXXXX		XXXXXXXX		xxxxxxxx	xxxxxxxx	30h
Reserved						34h
Reserved						38h
Max_Lat		Min_Gnt		Interrupt Pin	Interrupt Line	3Ch
Reserved				Retry Timeout Value	TRDY Timeout Value	40h
Reserved						44h-FFh

Table 4 *PCI Configuration Registers*

The default values for the above registers are as follows:

Device ID	= 2F44h	Subsystem ID	= 0000h
Vendor ID	= 11D4h	Subsystem Vendor ID	= 0000h
Class Code	= FF0000h	Max_Lat	= 00h
Revision ID	= 00h	Min_Gnt	= 00h
BIST	= 00h	Interrupt Pin	= 01h
Header Type	= 00h		

The upper 15 bits of the Base Address registers are writable, allowing the selection of a 128k-byte address space for the *CryptIC*. As part of automatic (plug & play) PCI address mapping, it is common for the Host BIOS to write FF's into the Base Address registers and then to read-back the value to determine the address range required by the target PCI device. In the case of the *CryptIC*, the lower 17 bits will be read as 0's, indicating 128k. Then, the BIOS writes an appropriate Base Address into the upper 15 bits which were read as 1's.

2183 IDMA Host Processor Bus

The 2183 IDMA Host selection (Internal Direct Memory Access) allows the *CryptIC* to offer the IDMA interface directly to an outside Host processor. The *CryptIC*'s usage of the IDMA bus is identical to that described in the ADSP 2183 datasheet and ADSP-2100 Family User's Manual.

The IDMA port allows a Host Processor to perform 16-bit DMA reads and writes of selected areas of the *CryptIC*'s internal memory space. These areas include: Internal Data Memory (DM) and Internal Program Memory (PM). Since PM is 24-bits wide, two IDMA cycles are required to access it. IDMA transfers are implemented using cycle-stealing from the *CryptIC*'s internal DSP processor. Note that the *CryptIC* supports optional memory locking of 1kword slices of DM or PM. Any locked areas of memory are not visible to a Host via the IDMA port. Typically, the locking of these memory spaces is performed by a custom 'Extended' program invoked via the CGX Kernel interface.

External Memory Interface

The External Memory Interface (EMI) bus is a logical extension to the EMI bus presented on a standard ADSP 218x processor. The *CryptIC* has enhanced this bus as follows:

- Extended data bus width from 8 / 16 / 24-bits to 8 / 16 / 24 / 32-bits
- Additional Addressing: from 14-bits (16k words) to 26-bits (64M words)

The EMI interface can support multiple memory types, including I/O, Program Memory (PM), Byte Memory (BM), and 16-bit or 32-bit Data Memory (DM).

Since the EMI bus is shared between the DSP and the DMA engine within the *CryptIC*, a control register is used to select which bus controller 'owns' the EMI bus. (This applies to all EMI accesses – PM, DM, I/O, BM). Only the DSP has access to this register, so it effectively becomes the arbiter for the EMI bus. This allows straightforward contention management between direct DSP access and DMA access

to external memory. It becomes more interesting when Host-initiated Target transfers are expected to external memory, since there is no intrinsic arbitration provided. If a Target transfer is attempted while the DSP owns the EMI bus, the PCI transfer will re-try and then be aborted.

5

32-BIT DMA CONTROLLER

Overview

The *CryptIC* integrates a high-performance 32-bit DMA controller in order to facilitate bulk data movements within the chip without requiring continuous DSP supervision. The DMA subsystem allows 32-bit transfers to occur within the *CryptIC* at up to 40 Mwords per second (160 Mbytes/s).

10

Figure 6 illustrates the functionality of the 32-bit DMA subsystem.

DMA Controller Functional Description

15

The DMA controller is shared between one of two 'owners'; either the DSP or the PCI Host processor. This essentially corresponds to whether a 'Master' (DSP-owned) or 'Target' (Host-owned) transfer is needed. An arbiter manages any contention issues when both the DSP and the Host attempt to control the DMA engine at the same time, with Target transfers getting priority.

20

All DMA operations occur on 32-bit buses within the *CryptIC*, although for some internal locations, the source or destination could be 16-bits wide. In this case, a bus interface state machine converts the data between 16 and 32 bits (see the '*' markings in the figure above).

25

Because the External Memory bus interface is also multiplexed between the DMA engine and a direct connection to the DSP's EMI bus, a DSP-controlled register output bit (PCIEXTMEM bit 1) selects which is the 'owner' of the External Memory bus. In typical applications (and the CGX Kernel), this bit is normally set to give control to the DMA engine – ensuring that Target bus transactions can complete – and would only be momentarily switched to the DSP during a non-DMA EMI transfer.

DSP Initiated Transfers

When the DSP controls the DMA engine, it truly behaves as a general-purpose DMA controller: The DSP specifies source and destination devices/addresses and the byte count, and the DMA engine then executes the transaction. Status registers may be polled for completion, or an interrupt may be generated at the end of the transfer.

For a **PCI host bus**, data movements can be handled between:

Case 1. PCI Host <—> DSP/Crypto*

* DSP/Crypto includes: All Crypto registers, Hash/Encrypt block, IDMA to DSP internal RAM, and Kernel RAM – if unlocked.

Case 2. PCI Host <—> External Memory

Case 3. External Memory <—> DSP/Crypto*

For cases 1 & 2 above, this is how a PCI 'Master' transaction occurs. Case 3 is a memory-to-memory type transfer.

For a **PCMCIA host bus**, data movements can be only handled as Target reads/writes between:

Case 1. PCI Host <—> DSP/Crypto*

For most DMA transactions, both the source and destination address pointers will be automatically incremented for each word transferred. The only exceptions to this are when either the source or destination of a transfer is:

- Hash/Encrypt Input FIFO
- Hash/Encrypt Output FIFO
- IDMA Data Register (Indirect mode only)

For those transfers, the FIFO/register address remains fixed and only the 'memory' side address automatically increments.

Memory Map

Addresses and memory domains are specified as part of the DMA setup. Each of the 3 domains – Host memory, External Memory, and DSP/Crypto-register – have slightly different addressing techniques as shown below.

5

Host memory:

When one end of the DMA transaction is PCI memory (case 1 & 2 above), a full 32-bit address is specified in the Host Address register. This allows a transfer to be done to/from any location in PCI memory space.

10

External memory:

When transferring to or from External memory, the addressing is slightly different for cases 2 & 3 above. For case #2 (PCI Host to/from External memory), a full 26-bit external memory address is specified in the Local Address register. Also, bit #31 in the Local Address register is set to '1' to indicate External Memory.

15

For case #3 (External memory to/from DSP/Crypto-registers), bit #14 in the Command register is set to '1'. Then, the 26-bit external memory address is specified in the Host Address register.

20

DSP/Crypto-register memory:

When one end of the transaction is DSP/Crypto-register/KRAM space, then a Word (16-bit) address is specified in the Local Address register. This causes the address map to be shifted one address bit from the PCI (byte-oriented) map shown in Figure 7 is the word memory map of DSP/Crypto-register addresses.

25

DMA Control Flow

Figure 8 is a flow chart showing the steps which are typically followed for a DSP-initiated Master transfer.

Code Samples

Below are some code samples for the three types of DSP-initiated DMA transfers:

1. PCI Host <--> DSP/Crypto

```

5  |void
|pci_master_transfer(unsigned short *host_hiaddr, - Host 31-16 address bits
|               unsigned short *host_loaddr, - Host 15-0 address bits
|               unsigned short *dsp_addr, - DSP 15-0 address bits, intnl
|               unsigned short read, - 0=write, non0 is a read
|               unsigned short bytes) - Number of bytes to read/write
10 |               of Host memory
|
| This operation is used to read/write N bytes (i.e. bytes) from/to PCI
| HOST memory using the DMA controller and performing master reads/writes.
|
15 | The PCI HOST addresses are the source addresses. In this example, the
| DSP is the destination. In fact, the DSP address will be to a crypto
| register. This code is not concerned with locking in external memory,
| so it will not allow external memory access.
|
20 | This operation will not bump the Host or DSP addresses after the read/write
| of the block is complete. This will be left to the calling operation.
|
| Register ar contains the 1st argument: host_hiaddr, and register ay1 contains
| the 2nd argument: host_loaddr. This is how the calling interface works with
25 | Analog Device's C compiler and assembler tools.
|
| This operation assumes the calling function will not request a read or a
| write that exceeds the maximum transfer size of 1023 bytes.
|
30 |pci_master_transfer_:
|   i1=i4; !get ptr to stack to get remaining args
|   m3=1;
|   modify(i1,m3); !align to argument 3, dsp_addr
|   my1=dm(i1,m3); !obtain dsp address, dsp_addr
35 |   mr0=dm(i1,m3); !obtain transfer direction, read
|   sr0=dm(i1,m3); !obtain byte cnt, bytes
|   si=dmovlay; !save away old DMOVLAY value
|   dmovlay=15; !enable CryptIC register's data page
|
40 | ! Ensure that the DMA controller is available to accept a command.
|
|dma_control_ready:
|   sr1=dm(0x1845); !read DMA status
|   af=sr1 and 0x8000; !if DMA control is not ready
45 |   if eq jump dma_control_ready; !then continue to poll for ready
|
| ! Setup DMA address block for actual PCI HOST master read operation
|
|   dm(0x1840)=ay1; !lower 16 bits of 32bit address of host

```

5	<pre> dm(0x1841)=ar; !upper 16 bits of 32bit address of host dm(0x1842)=my1; !DSP address, 0:15 sr1=0; dm(0x1843)=sr1; !internal access, no upper address ! ! Setup the DMA command and issue it: ! No HOST generated interrupt </pre>
10	<pre> ! Byte count as read from stack above this ! Master read or write, depends on read argument ! PCI transfer ! ar=sr0; !sets up transfer as master read af=pass mr0; !if transfer is a read if ne jump dma_write_control; !then write DMA control register ar=sr0 or 0x8000; !set master write bit with length </pre>
15	<pre> dma_write_control: dm(0x1844)=ar; !write DMA control reg. ! ! Important, insert a delay of 2 instruction cycles to ensure that ! the transfer-active flag of the DMA status register will have time ! to be asserted. This avoids a false read by the DSP. ! nop; nop; ! ! Poll until the PCI transfer is complete ! </pre>
20	<pre> dma_master_transfer_poll: sr1=dm(0x1845); !read DMA status af=sr1 and 0x0008; !if master transfer in progress if ne jump dma_master_transfer_poll; !keep polling ! ! Restore the DMOVLAY register and return to the calling function. ! </pre>
25	<pre> dmovlay=si; !restore previous data page rts; !return to caller, pop return address </pre>
30	
35	

PCI Host Initiated Transfers (Target Mode)

When a PCI Host performs a Target Read or Target Write of memory or a register within the *CryptIC*, the DMA controller is automatically called into use. From the Host's perspective, most of the operation of the DMA controller is hidden. The DMA controller interprets the PCI-supplied addresses and other bus control signals and then generates the appropriate addresses for internal/external memory space.

For some PCI Target reads which experience latencies before data is retrieved, the DMA controller may 'fetch-ahead' two or three dwords and place them in the PCI

core's read FIFO. This is important to consider when performing reads from the Hash/Encrypt FIFO, since once data is read from the H/E FIFO, it cannot be reversed. Thus, the Host must ensure that no 'fetch-ahead' is performed in these cases.

5 The PCI_Target_Read_Count register is the mechanism to limit the maximum 'fetch-ahead' data which can be read. For example, if the Host is moving data through the Hash/Encrypt FIFOs in 8-dword blocks, then the PCI_Target_Read_Count register should be programmed with an 0x08. See section on Target Read Count Register (TARGRDCNT) described further below.

10 All Target mode reads or writes must occur on 32-bit dword boundary. The only exception is reads or writes to the Hash/Encrypt block. In this case, the PCI starting and ending address is decoded down to the byte level so that any number of bytes may be written to or read from the H/E data FIFO.

PCMCIA Host Initiated Transfers

15 All PCMCIA transactions to the *CryptIC* are Target transfers. The DMA controller described in this chapter is disabled when the PCMCIA bus is selected. Instead, a separate 16-bit DMA controller is enabled and is controlled via the PCMCIA DMA controller registers described in Applications Registers.

Both single-word reads and writes, as well as multi-word burst DMA transfers are supported.

20 DMA Arbitration

Since the DMA is a shared resource between the DSP and Host Target mode requests, a deterministic arbitration scheme is required for predictable results. The Arbiter shown in Figure 6 is responsible for moving DMA requests into the DMA engines working registers. The Arbiter gives priority to Host Target mode requests, so this means the following:

25 Assume a DMA transaction is in progress, and that another DSP-initiated transfer request is queued-up in the DSP control registers. If a Host (PCI or PCMCIA) Target read or write request occurs before the in-progress transaction has

completed, then the Host request will be serviced prior to the queued DSP request. Note that any DMA transfer which is already running will never be preempted. The only way for a DMA transfer to be aborted in mid-stream is for a Host bus error to occur (e.g. a PCI abort due to a Parity error) or for the DSP to force an abort by writing to the PCICSC register 'Force End Transfer' bit.

DMA Register Set

A set of memory-mapped control and status registers are used to operate the DMA controller. These are considered *Unprotected Registers*, and therefore are visible to either the DSP running in User mode or to an outside PCMCIA/PCI bus entity. They are summarized in Table 5 and described in detail in the following subsections.

(In the table below, 16-bit address refers to the DSP and 32-bit address refers to PCI/PCMCIA host.

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset Default	DESCRIPTION
PCI APPLICATION REGISTERS					
DSP-Visible Registers:					
0x1840	N/A	Host Address [15:0]	R/W		Lower 16-bits of Host Address
0x1841	N/A	Host Address [31:16]	R/W		Upper 16-bits of Host Address
0x1842	N/A	Local Address [15:0]	R/W		Lower 16-bits of Local Address
0x1843	N/A	Local Address [31:16]	R/W		Upper 16-bits of Local Address
0x1844	N/A	Command	R/W		DMA Command
0x1845	N/A	Status/Config.	R/W		DMA Status/Configuration Register
0x1846	N/A	PCI Core Status/Config.	R/W		Status/Configuration for PCI core
0x1847	N/A	PCI Extmem Status	R		PCI External Memory Status
PCI Host-Visible Registers:					
N/A	0x00C0-C3	PCI Target Page	R/W		Target Page specifier when <i>CryptIC</i> External Memory is Target
N/A	0x00C4-C7	PCI Target Read Count	R/W		Maximum number of dwords for Target Read Transfer
N/A	0x00C8-CB	Endian	R/W		Big/Little Endian select
PCMCIA Host-Visible Registers:					
N/A	0x00C0-C1	DMA Address	R/W		DMA address register
N/A	0x00C4-C5	Start/Stop DMA	R/W		DMA control
N/A	0x00C8-CB	Endian	R/W		Big/Little Endian select

Table 5 DMA Controller Register Set

PCI Host Address Low Register (PCIHAL)

This 16-bit Read/Write register allows the DSP Software to configure the lower 16 bits of a PCI Host Address for a Master mode transaction. For a DSP-to-External memory transfer, this contains the lower 16-bits of the External Memory address, as shown in the table below. Note that this is a byte address.

Register Address (READ / WRITE)															
DSP				PCMCIA				PCI							
0x1840				Not Visible				Not Visible							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	x	X	X	x	x	x	x	x	x	x	x	x	X	x	x
Msb												lsb			

16 lsb's of Host Address [15:0]

PCI Host Address High Register (PCIHAH)

This 16-bit Read/Write register allows the DSP Software to configure the upper 16 bits of a PCI Host Address for a Master mode transaction. If the transfer is between External Memory and the DSP memory space (case 3), then this register holds the 10 most-significant bits of the External Address [25:16], as shown in the table below. Note that this is a byte address.

Register Address (READ / WRITE)															
DSP				PCMCIA				PCI							
0x1841				Not Visible				Not Visible							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x
Msb												lsb			

16 msb's of Host Address [31:16]

PCI Local Address Low Register (PCIHAL)

This 16-bit Read/Write register allows the DSP Software to configure the lower 16 bits of Local (*CryptIC*) Address for a PCI Master transaction, as shown in the table below. Note that this is a 16-bit word address.

5

Register Address (READ / WRITE)		
DSP	PCMCIA	PCI
0x1842	Not Visible	Not Visible

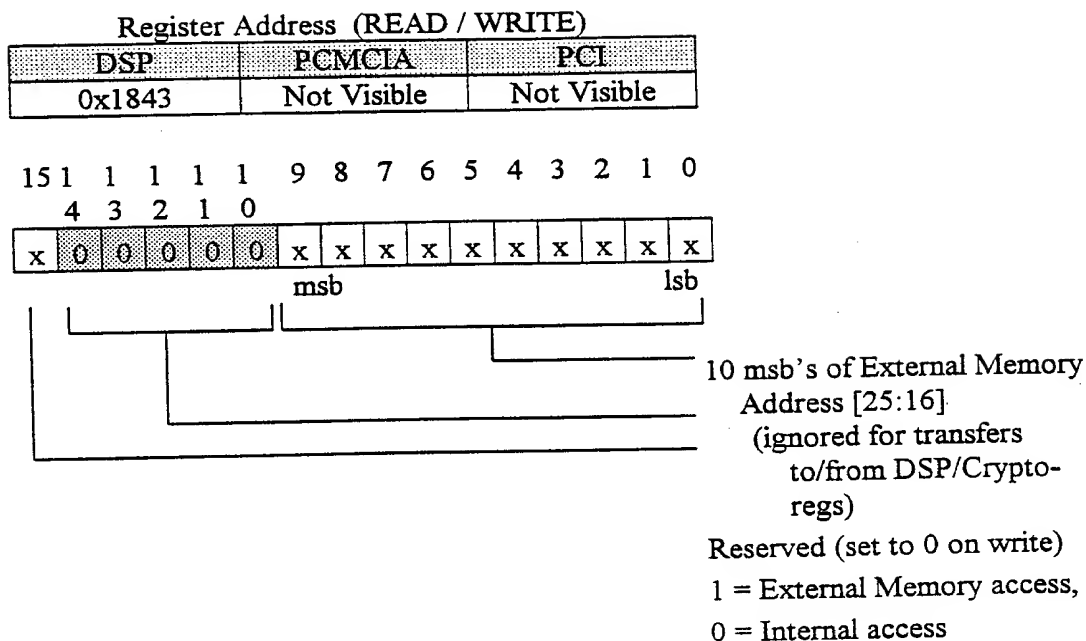
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
X	X	X	X	x	x	x	x	x	x	x	x	x	x	x	x	
Msb																lsb

16 lsb's of Local Address [15:0]

PCI Local Address High Register (PCIHAH)

If the transfer is between a PCI Host and External Memory (case 3), then this register holds the 10 most-significant bits of the External Address [25:16] and the most-significant bit will be set to '1'. If the transfer is to/from the DSP/Crypto-register space (cases 1 & 2), then there are no address bits contained here and the most-significant bit will be '0', as shown in the table below. Note that this is a 16-bit word address.

10

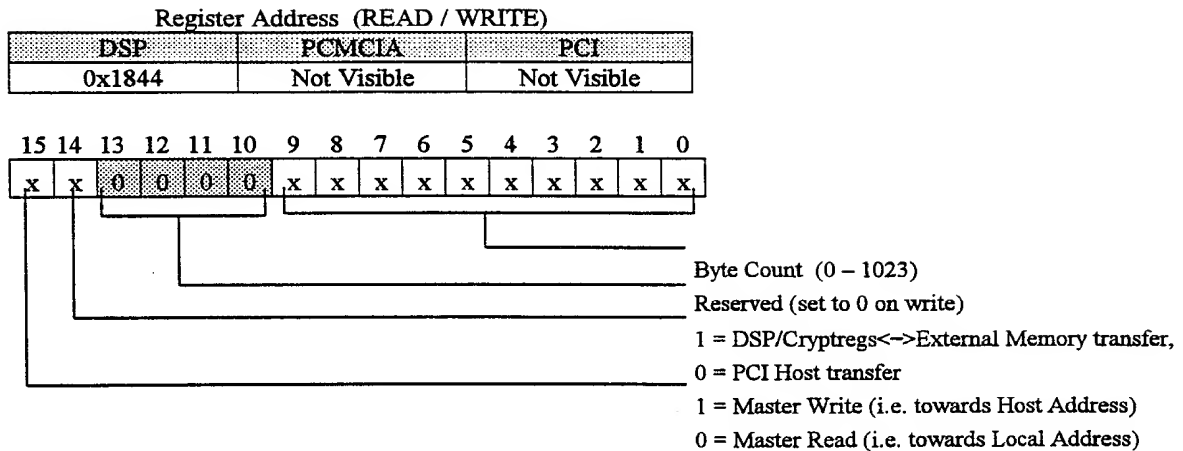


PCI Command Register (PCIC)

This 16-bit Read/Write register, as shown in the table below, is used by the DSP to write Commands to the DMA Controller function.

The first 10 bits indicate the byte count of the requested transfer. Bit 14 selects the type of transfer: Between the DSP/crypto registers and the External Memory space (case 3), or between a PCI Host and either External Memory or the DSP/crypto registers (case 1 or 2).

Bit 15 selects the direction of the DMA transfer.



DMA Status/Configuration Register (PCISC)

This 16-bit Read/Write register, as shown in the table below, allows the DSP to configure/monitor the DMA function.

The first 2 bits are Read/Write and select the Wait States when the DMA engine is transferring to or from External Memory. Note that the same number of Wait States should be selected internal to the DSP in the DWAIT bits of the Wait State Control Register.

Bit 2 is a Read-Only status bit which reflects the Host-selected Endian state. All memory and registers within the DSP are Little Endian. The Endian bit determines whether or not the *CryptIC* has to do Endian conversion on data to/from the host.

The next three bits [3-5] are general status bits which indicate the busy status of the DMA engine for each of its three modes:

- Bit 3 set to 1 indicates that a DSP-initiated master transfer is running (could be case 1, 2 or 3). Note that when this bit transitions from 1 to 0, it may cause a *Master PCI Transfer Complete* interrupt to occur (see section 0).
- Bit 4 set to 1 indicates that a Host-initiated target transfer is running (could be case 1 or 2).

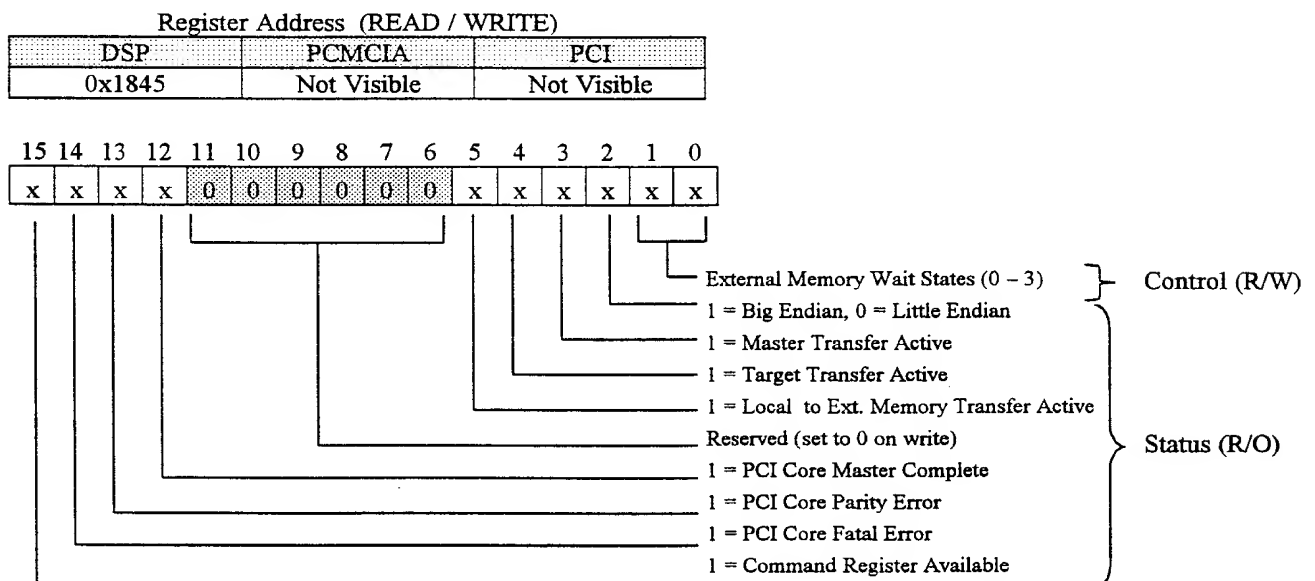
- Bit 5 set to 1 is a further qualifier on Bit 3 (i.e. bit 3 will also be set): It indicates that the transaction is for case 3.

Bits 12-14 provide PCI core status to the DSP:

- Bit 12 indicates that the PCI core has completed a DSP-initiated master transfer.
- Bit 13 indicates that the PCI core has detected a PCI parity error on the bus.
- Bit 14 indicates that the PCI core has experienced a PCI fatal error.

The last bit [15] indicates that the DSP may write into the DMA engine register set. (Note that another DMA transfer may be underway, but since the DSP side has double-buffered registers, another set of addresses and a command may be queued. Note that when this bit transitions from 0 to 1, it may cause a *Master PCI Transfer*

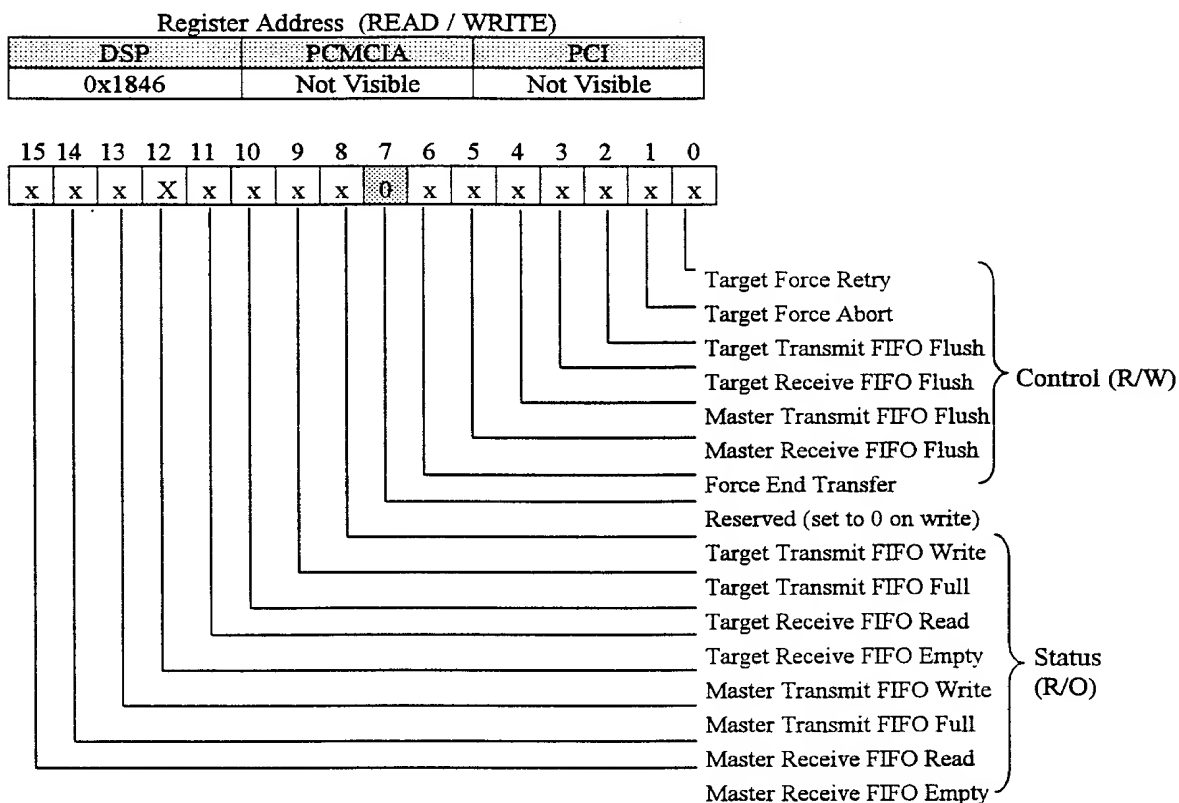
Queued interrupt to occur. (See section on DSP Unmasked Status Register (DUSTAT) described further below.)



PCI Core Status/Configuration Register (PCICSC)

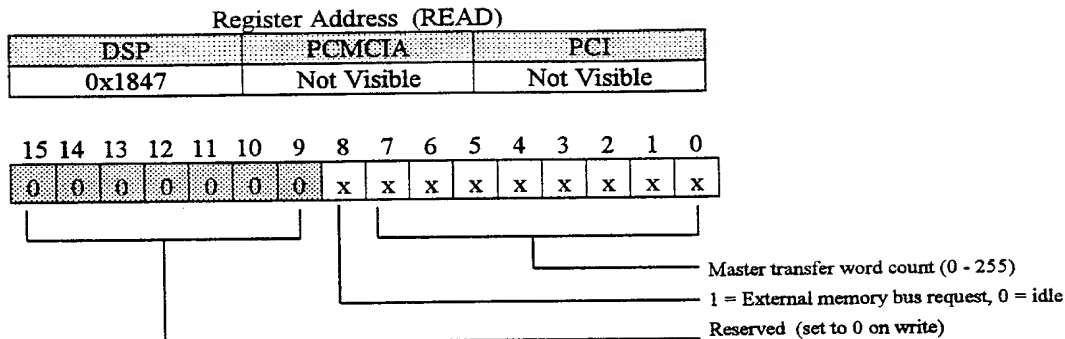
This 16-bit Read/Write register, as shown in the table below, allows the DSP to configure and monitor the PCI Core function. This register is not normally accessed for most applications.

The first 7 bits allow the DSP to terminate PCI transfers under abnormal circumstances. The last 8 bits provide real-time visibility of PCI core operation status.



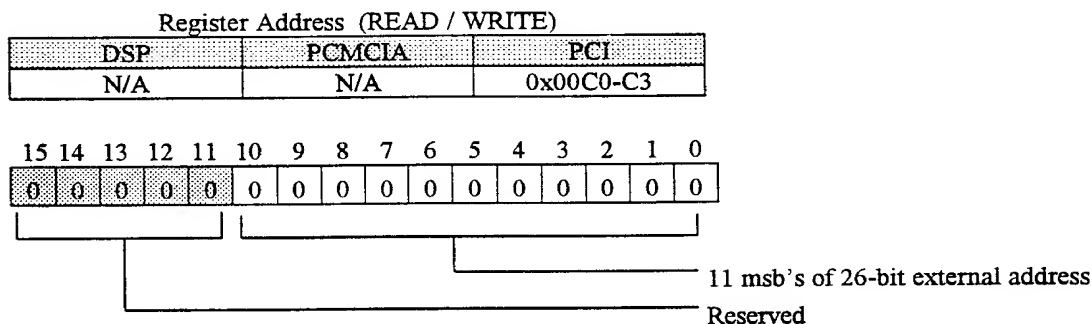
PCI External Memory Status Register (PCIEMS)

This 16-bit Read only register, as shown in the table below, reports the status of External memory and Master transfers. The least-significant 8 bits report on the current word count of a transfer. They will start initialized to the number of words in the transfer and will decrement down to 0. Bit 8 indicates if the External Memory bus is in use by the DMA engine.



PCI Target Page Register (TARGPAGE)

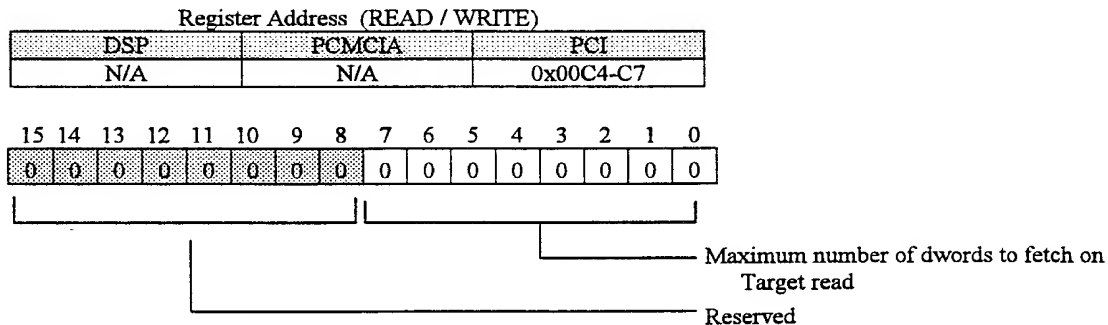
The table below shows the bit definitions for the Target Page Register. These are used in order to select the 64 kbyte page which the PCI Host may access for a Target read or write. This register is not used for DSP-initiated (Master) transfers. Note that this register is only visible to the PCI Host processor.



Target Read Count Register (TARGRDCNT)

This register, as shown in the table below, specifies the maximum number of dwords to fetch after a Target mode read has begun. Since Target reads can sometimes timeout due to the access latencies in the path from PCI core to the addressed location, it is desirable to fetch enough data so that on the PCI re-try, sufficient data will be available in the PCI core read FIFO to complete the transaction.

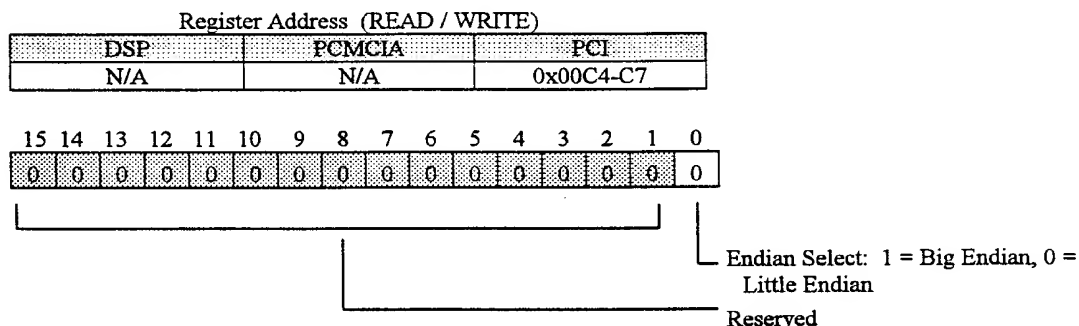
On the other hand, anticipatory fetching data from an internal FIFO such as the Hash/Encrypt data FIFO can be dangerous. If the Target read only requires 2 bytes from the FIFO, and 8 bytes are pre-fetched, then data will be lost. For Target reads of the FIFOs, this register should be programmed with the size of the transfer.



Endian Register (PCIENDIAN)

This register, as shown in the table below, specifies the 'Endianness' of data transfers between the PCI bus and the *CryptIC*. The DSP is little-endian, so if it is communicating with a big-endian Host, then byte swapping is needed. Setting a 1 in this register will cause a hardware byte-swap to occur on all PCI transfers to any element of the *CryptIC*, including external memory and internal registers or memory spaces.

The status of this Host selection is reflected on the DSP side in the DMA Status/Configuration register, described above.



HASH/ENCRYPT SUBSYSTEM

Hash and Encrypt Block Overview

The Encrypt Block is tightly coupled to the Hash Block in the *CryptIC* and therefore the two are discussed together. Refer to Figure 9 for the following description:

The algorithms implemented in the Combined Hash and Encryption Block are: DES, Triple DES, MD5 and SHA-1. Data can be transferred to and from the module once to perform both hashing and encryption on the same data stream. The DES encrypt/decrypt operations are highly paralleled and pipelined, and execute full 16-round DES in only 4 clock cycles. The internal data flow and buffering allows parallel execution of hashing and encryption where possible, and allows processing of data concurrently with I/O of previous and subsequent blocks. Context switching is optimized to minimize the overhead of changing cryptographic keys to near zero.

The 'software' interface to the module consists of a set of memory-mapped registers, all of which are visible to the DSP and most of which can be enabled for Host access. A set of five, 16-bit registers define the operation to be performed, the length of the data buffer to be processed, in bytes, the offset between the start of hashing and encryption (or vice versa), and the Padding operation. If the data length is unknown at the time the encrypt/decrypt operation is started, the Data Length register may be set to zero which specifies special handling. In this case, data may be passed to the Hash/Encrypt block indefinitely until the end of data is encountered. At that time, the operation is terminated by writing a new control word to the Hash/Encrypt Control Register (either to process the next packet or to invoke the 'idle' state if there is no further work to do). This will 'close-out' the processing for the packet, including the addition of the selected crypto padding.

A set of seven status registers provides information on when a new operation can be started, when there is space available to accept new data, when there is data available to be read out, and the results from the Padding operation.

Crypto Contexts

There are also two sets of 'crypto-context' registers. Each context contains a DES or Triple DES key, Initialization Vector (IV), and pre-computed hashes (inner and outer) of the Authentication key for HMAC operations. The contexts also contain registers to reload the byte count from a previous operation (which is part of the hashing context), as well as an IV (also called 'salt') for decrypting a Black key, if necessary.

Once a crypto-context has been loaded, and the operation defined, data is processed by writing it to a data input FIFO. At the I/O interface, data is always written to, or read from, the same address. Internally, the hash and encryption functions have separate 512-bit FIFOs, each with their own FIFO management pointers. Incoming data is automatically routed to one or both of these FIFOs depending on the operation in progress.

Output from the encryption block is read from the data output FIFO. In encrypt-hash or decrypt-hash operations, the data is also automatically passed to the hashing data FIFO. Output from the hash function is always read from the digest register of the appropriate crypto-context.

The Initialization Vector (IV) to be used for a crypto operation can be loaded as part of a crypto-context. When an operation is complete, the same context will contain the resulting IV produced at the end, which can be saved away and restored later to continue the operation with more data.

In certain packet-based applications such as IPsec, a feature is available that avoids the need to generate and load random IV's for outgoing (encrypted) packets. The operating sequence for this feature is as follows:

- 1) For the first encrypted packet after the *CryptIC* is initialized, two random numbers should be generated and written to each context's IV register. (This can actually be done as part of the *CryptIC* boot process.)
- 2) Control bit 0 in the Hash/Encrypt Control register is set to a '1' in order to prevent subsequent software overwriting of the IV field in the two context registers.
- 3) Now, at the end of each packet encryption or decryption, the IV register in the active context will contain the last 8 bytes of ciphertext. This 'random' value will remain in the IV register and not be overwritten when the context for the next packet is loaded. (This technique is fully compliant with the IPsec standards.)
- 4) For decrypted packets, the IV is typically explicitly included with the incoming packet. Thus, the Control bit in step 2 will have to be set to a '0' prior to writing the IV into the context register. After the IV is written, the control bit should be restored to '1'.

Padding

When the input data is not a multiple of 8 bytes (a 64-bit DES block), the encrypt module can be configured to automatically append pad bytes. There are several options for how the padding is constructed, which are specified using the pad control word of the operation description. Options include zero padding, pad-length character padding (PKCS #7), incrementing count, with trailing pad length and next header byte (for IPsec), or fixed character padding. Note that for the IPsec and PKCS#7 pad protocols, there are cases where the padding not only fills-out the last 8-byte block, but also causes an additional 8-byte block of padding to be added.

For the Hash operations, padding is automatically added as specified in the MD-5 and SHA-1 standards. When the 'Hash Final' command is issued indicating the last of the input data, the algorithm-specified padding bits are added to the end of the hash input buffer prior to computing the hash.

Data Offsets

Certain security protocols, including IPsec, require portions of a data packet to be Hashed while the remainder of the data is both hashed and encrypted. The *CryptIC* supports this requirement through the OFFSET register which allows specifying the number of 32-bit dwords of offset between the hash and encrypt operations.

Black Key Loads

The cryptographic keys loaded as part of a crypto-context can be stored off-chip in a "black" (encrypted) form. If the appropriate control bit is set (HECNTL bit 15), the DES or 3DES key will be decrypted immediately after it is written into the Context register. The hardware handles this decryption automatically. The KEK that covers the black keys is loaded in a dedicated KEK register within the *CryptIC*. The IV for decrypting the Black secret key is called 'Salt' and must be stored along with the black key (as part of the context). Note that 3DES CBC mode is used for protecting 3DES Black keys and single-DES CBC is used for single-DES Black keys.

When Black keys are used, there is a 6-cycle overhead ($0.18\mu\text{s}$ @ 33MHz) for DES keys or 36-cycle overhead ($1.1\mu\text{s}$ @ 33MHz) for triple-DES keys each time a new crypto-context is loaded. (Note that if the same Context is used for more than one packet operation, the Key decryption does not need to be performed again.) Depending on the sequencing of operations, this key decryption may in fact be hidden (from a performance impact perspective) if other operations are underway. This is because the Black key decryption process only requires that the DES hardware be available. For example, if the DSP is reading the previous Hash result out of the output FIFO, the Black Key decryption can be going on in parallel. Also note that the data driver firmware does NOT have to wait for the key to be decrypted before writing data to the input FIFO. The hardware automatically waits for the key to be decrypted before beginning to process data for a given packet. So it is possible to make the impact of black key essentially zero with efficient pipeline programming.

The Key Encryption Key (KEK) for key decryption is loaded via the Secure Kernel firmware using one of the CGX Key Manipulation commands (see “CGX Interface Programmer’s Guide”). This KEK is typically the same for all black keys, since it is usually protecting local storage only. It is designated the KKEK in the CGX API.

One of the Laser Programmed configuration bits specifies whether Red (Plaintext) keys are allowed to be loaded into the *CryptIC* from a Host. If the RedKeyLoad Laser bit is set, keys may *only* be loaded in their ‘Black’ form. This is useful in systems where export restrictions limit the key length which may be used or where the external storage environment is untrusted. If the BlackKeyLoad bit is *not* set, then keys may either be loaded either in their Black form, or in the ‘Red’ (unencrypted) form. Note that the Laser Configuration bit may be overridden with a signed Enabler Token (see “CGX Interface Programmer’s Guide”).

Depending on the definition of the ‘Security Module Boundary’ in a given application, FIPS 140-1 may require the use of ‘black key’ to protect key material. In other words, if the Security Boundary does not enclose the database where keys are stored, then those keys must be protected from compromise. Black key is a satisfactory way to meet this FIPS requirement.

Encrypt and Hash Detailed Description

The following sections provide details on the operation of the Hash/Encrypt block of the *CryptIC*.

DES Subsystem

The 512-bit (64 byte) crypto data FIFO allows up to eight 64-bit blocks to be queued for processing. The FIFO is implemented as a circular buffer, where the processed data is written back to the same location it came from. For most applications, the optimum transfer size is 256-bits (32 bytes) which provides the most

efficient 'pipelining'. This allows a set of four 64-bit blocks to be queued, then while those are being processed, the previous four blocks can be output, and a new set of four blocks input.

DES Modes

The *CryptIC* DES/3-DES engine can perform any of the standard DES modes: ECB, CBC, OFB, CFB in either Single-DES or Triple-DES. Since DES operates on 64-bits at a time, data is always input to the algorithm in 8-byte lumps (or padded to 8 bytes by the *CryptIC*). The mode of DES operation is selected in the HECNTL register at the same time as an Encrypt/Decrypt operation is started.

In the CBC, OFB and CFB modes, the DES algorithm block is used to generate a 'Keystream' which is XORed with Plaintext data in order to product Ciphertext. This XORing is performed within the Encryption hardware, so the user only passes Plaintext or Ciphertext data in and out of the *CryptIC*.

For Cipher Feedback mode, one of three feedback choices is available: 64-bit, 8-bit, or 1-bit. In 64-bit CFB, data is written to the input FIFO in the same manner as for all other modes. However in 8-bit or 1-bit CFB modes, null bytes must be written to the data FIFO in order to align the desired byte or bit within the 8-byte DES output. For example, in 8-bit CFB mode, 7 null bytes must be written after each 'payload' byte is written to the input FIFO.

The Triple-DES (3DES) processing performed by the encrypt hardware employs the '*Outer*' 3DES algorithm (as opposed to '*Inner*' 3DES). This means that, for a given input block of 8-bytes, the DES engine is first run in Encrypt, then Decrypt, and finally again in Encrypt mode prior to any feedback or XOR operations. *Inner* 3DES performs feedback operations between each of the 3 DES operations. Most of the security protocol standards call for *Outer* 3DES, and it is considered the stronger of the two modes.

Crypto Padding

To facilitate peak encrypt/decrypt performance, the *CryptIC* supports the most commonly needed Padding functions in hardware. The features include:

- Generating and appending Pad bytes to the end of a Plaintext packet prior to encryption
- Verifying correct Pad bytes after decrypting a packet
- Consuming (discarding) Pad bytes after decrypting a packet.

Four Padding Modes are supported in the *CryptIC* hardware, as shown in the table below.

Number	Mode	Description
0	Zero Pad	Appends 0 to 7 bytes of 0x00 to the Plaintext data to ensure the total number of bytes has no remainder modulo 8.
1	IPsec	Appends 0 to 7 pad bytes, followed by pad count 'n' (0 to 7) and then a "next header" byte. The pad byte values are a count from 1 to n. The 'Next Header' byte is specified in the Pad Control register. A total of 2 to 9 bytes may be appended.
2	PKCS #7	Appends 1 to 8 bytes: Pad byte value = hex value of Pad count, so if 3 pad bytes are needed, they will be: '03, 03, 03'.
3	Constant Pad	Appends 0 to 7 bytes of a user-specified character to the Plaintext data to ensure the total number of bytes has no remainder modulo 8. The byte (any value from 0x00 to 0xff) is specified in the Read/Write <i>Pad Control Register</i> .

If the Host system software wishes to implement another type of Padding than is supported in hardware, then Mode 0 (Zero Pad) should be selected. The Host simply insures that the end of the data to be encrypted falls on an 8-byte boundary by inserting Pad characters on its own, in which case the Hardware Padding engine will not add any bytes.

Pad Verification

There is a Pad Verify bit in the General Status register which checks for proper padding in Pad Modes 1 and 2. (Note that this bit is invalid for all blocks read from the hash/encrypt FIFO except the last block to be processed.) The Pad Verification

operation checks the decrypted data for the correct Pad properties as specified in the selected *Padding Mode* (The *Next Header* byte value is not validated for IPsec mode). If Pad Modes 0 or 3 are selected, the Pad Verify bit will always read 0.

Pad Consumption

The application must always read-out the last block (8-bytes) of decrypted plaintext data if there is at least one user-payload byte in it. When either Pad Mode 1 (IPsec) or Pad Mode 2 (PKCS #7) is selected, the *CryptIC* can notify the application of the number of Pad bytes (including Pad, Pad length, and Next Header if applicable) detected in the decrypted plaintext through the Pad Status registers (HEPADSTAT0/1). A count of 0 to 9 can be reported.

In addition, these padding modes can cause an additional block of 8 bytes to be produced (since more than 7 bytes may have originally been added to the packet). The presence of this additional block is detected by the *CryptIC* and the application may command the *CryptIC* to discard the last Pad block in order to save the time of reading those 8 bytes. A write of any value to the Consume Pad register will cause this discard to occur.

Hash Subsystem

Like the Encrypt/Decrypt subsystem, the hash processing section also has a 512-bit (64 byte) FIFO. This represents a single 512-bit input block to the hash algorithm. When the hash buffer is full, and the algorithm section is done processing previous data, the input block is immediately copied to the 512-bit algorithm working buffer, and the entire 512-bit FIFO buffer is available for data input again.

There are several options available to control hashing, in addition to selecting between SHA-1 and MD5. The application may choose to either set the initial state of the hash operation from the constants defined for the algorithms, or from a digest which is loaded as part of a crypto-context. To continue a previous operation, the

previous 'interim' digest must be reloaded. If an operation is to be resumed, it is also necessary to load the previous count of bytes processed. This can also be loaded as part of the crypto-context.

Hash Padding

Controls are also provided to determine what is done at the end of the input data stream. If the operation is to be resumed at a later time, then the operation should be defined to exclude 'final' processing. In this case, no special processing will be done at the end beyond making the resulting digest available as part of the output crypto-context. If the operation is to include the end of last of the input data, then the control for 'final' processing should be set, which will cause the padding operations defined for the hashing algorithms to be performed. These include adding a 'pad byte' following the last byte of data, and placing a 64-bit data length, expressed in terms of bits, at the end. If less than 9 bytes are available between the last input data byte, and the next 512-bit block boundary, an extra 512-bit padding hash block will be added to contain the length. Any additional bytes required to fill out the last or extra 512-bit blocks are set to zero.

As previously described, the same input data stream is internally buffered by both the hash and encryption sections, depending on the data flow of the operation selected. In the case of hash-encrypt, where the two components of the operation are done in parallel, if any padding is added to the crypto block according to the option selected, the same padding is added to the hash block.

HMAC

To support the IETF HMAC protocol, the module supports processing an 'outer' hash. Each crypto-context supports loading an 'inner' and an 'outer' pre-computed initial digest. Typically, these would be the results of the HMAC keyed hash pre-processing, and would be stored as part of the security association (SA) negotiated for an IP connection. The pre-computed inner hash is loaded as the initial state of the hash algorithm before processing the input data stream. At the end of the

input data, normal 'final' processing is done, then the resulting digest is used as data for an additional round of hash processing, where the initial state is the pre-computed initial 'outer' keyed hash digest.

Hash/Encrypt Data Flow

Depending on how IP headers are processed in the packets to be transformed using hash-encrypt, it may be that the data buffer ends up not being double word aligned with respect to the starting point of the operation in the buffer. If this is the case, PCI transfers can be done using PCI byte enables to begin and end the operation on arbitrary bytes within their respective double words. When the on-chip DSP is used to setup the PCI transfers using PCI Master Mode, it has the capability to control the byte enables by setting the starting offset and byte count. When PCI Target Mode transfers are used, the PCI host is responsible for controlling the byte enables. Byte enables are also used on the 16-bit DSP bus. In this case, the enable signals are set in a register in another block and passed in as a two bit wide signal. This capability only applies to I/O to the data FIFOs. All other registers assume full word (16-bit bus) or double word (32-bit bus) transfers.

The hash-encrypt block also supports communication with host processors which may be either big endian or little endian, in terms of the order of storage within a double word. The default assumption is little endian. If big endian is selected, the module will reorder the input and output bytes, and in the case of the 32-bit bus, align them with the proper edge of the resulting double word. Endian processing applies to both the data and crypto-context registers, but not the control and status registers, where it can be handled in the application software.

The hash-encrypt block is designed to support zero wait state reads on the 16-bit bus to the DSP, and 1 wait state reads on the 32-bit bus to PCI (due to endian processing). Writes experience a one clock latency, as they are first latched, then written to the target address. Writes can be zero wait from the host's perspective, however, as long as a read is not attempted in the immediately following cycle.

Hash/Encrypt Subsystem Notes

Note that due to the pipelining nature of the module, it is possible to begin a new operation before all the data from the previous operation has been processed and read out. The sequence of steps is simplified if operations are performed serially with no overlap.

Final processing for hashing operations, and padding of a short trailing crypto block, is initiated automatically. The end of the input data stream for an operation is determined by counting the number of bytes input and comparing to the byte length entered as part of the operation control. An operation may be ended prematurely by entering a new operation control. If the length field is set to zero, this is the only way to cause a normal end.

Operations can be aborted by resetting the hash-encrypt block. All registers except the KEK will be reinitialized. The hash-encrypt block can be reset in the *CryptIC* via the kernel mode block hash-encrypt control register.

If black keys are used, the KEK register must be loaded prior to loading either of the crypto contexts.

In the case where the DSP is performing all hash-encrypt actions as a result of calls from application software to the Crypto CGX interface, then all actions described above are reads and writes via DSP (16-bit).

When the DSP is managing the interactions in PCI system, and the data resides in PCI host memory, then the status polling and control actions take place via DSP reads and writes on the 16-bit bus. In this case, however, the crypto-context and data FIFO I/O takes place on the 32-bit bus, as a result of PCI Master Mode transfers initiated by the DSP.

When a host processor is directly managing all aspects of hash-encrypt operation, then all I/O takes place on the 32-bit bus, although in some cases the upper

16-bits of the bus are not used (such as in a PCMCIA environment or with a 16-bit embedded processor).

The Kernel Mode firmware releases access of the Hash/Encrypt function block to the PCI/PCMCIA host whenever it is not executing a CGX command.

Hash/Encrypt Registers

Table 6 lists the memory-mapped register interface to the hash-encrypt module within the *CryptIC*. These registers may be accessed either from the DSP in Kernel mode, or if granted permission, from a PCI/PCMCIA host external to the *CryptIC*. Addresses on the 16-bit bus from the DSP are to 16-bit words. Addresses on the 32-bit bus (from PCI host) are to the first 8-bit byte of the 32-bit transfer. Transfers on the 32-bit bus should always be aligned to double word boundaries.

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset Default	DESCRIPTION
Configuration Registers:					
0x1000	0x0200	H/E control	R/W	0x0000	Hash/Encrypt block control word
0x100f	0x021e	Pad control	R/W	0x0000	Pad control word
0x1010-1011	0x0220-0223	Length	R/W	0x0000	32-bit data length, in bytes
0x1012	0x0224	Offset	R/W	0x0000	Offset, (0 to 15 in dwords), from start of hash (encryption) to start of encryption (hash)
0x1013	0x0226	Control	R/W	0x0007	Operation control.
0x1014	0x0228	Consume Pad	W		Command to consume final pad block
Status Registers:					
0x1016	0x022c	Pad Status 0	R		Decrypted next header byte, # Pad bytes, Context 0
0x1017	0x022e	Pad Status 1	R		Decrypted next header byte, # Pad bytes, Context 1
0x1018	0x0230	GeneralStatus	R		Status result from Hash/Encrypt operation
0x1019	0x0232	ControlReady	R		1 = input ready for new control word, 0 = not ready
0x101a	0x0234	DataReady	R		1 = input ready for data FIFO, 0 = not ready
0x101b	0x0236	StatFreeBytes	R		Number free input bytes in crypto FIFO (in 64-bit blocks)
0x101c	0x0238	StatOutBytes	R		Number output bytes ready in crypto FIFO (in 64-bit blocks)
Context 0 Registers:					
0x1020-1023	0x0240-0247	Key3_0	W		Key 3, for Triple DES: Crypto Context 0
0x1024-1027	0x0248-024F	Key2_0	W		Key 2, for Triple DES: Crypto Context 0
0x1028-102B	0x0250-0257	Key1_0	W		Key 1, for Triple DES or DES: Crypto Context 0
0x102C-102F	0x0258-025F	Salt_0	W		IV for key decryption: Crypto Context 0
0x1030-1033	0x0260-0267	IV_0	R/W		IV for data encrypt/decrypt: Crypto Context 0
0x1034-103D	0x0268-027B	Digest_0	R/W		(Inner) Digest: Crypto Context 0
0x103E-1047	0x027c-028F	OuterDigest_0	W		Outer Digest: Crypto Context 0

0x1048-1049	0x0290-0293	HashByteCnt_0	R/W		Starting byte count, for hash resume: Crypto Context 0
Context 1 Registers:					
0x1050-1053	0x02A0-02A7	Key3_1	W		Key 3, for Triple DES: Crypto Context 1
0x1054-1057	0x02A8-02Af	Key2_1	W		Key 2, for Triple DES: Crypto Context 1
0x1058-105B	0x02B0-02B7	Key1_1	W		Key 1, for Triple DES or DES: Crypto Context 1
0x105C-105F	0x02B8-02BF	Salt_1	W		IV for key decryption: Crypto Context 1
0x1060-1063	0x02C0-02C7	IV_1	R/W		IV for data encrypt/decrypt: Crypto Context 1
0x1064-106D	0x02C8-02DB	Digest_1	R/W		(Inner) Digest: Crypto Context 1
0x106E-1077	0x02D0-02EF	OuterDigest_1	W		Outer Digest: Crypto Context 1
0x1078-1079	0x02F0-02f3	HashByteCnt_1	R/W		Starting byte count, for hash resume: Crypto Context 1
Data In/Out FIFOs:					
0x10c0	0x0380	Data FIFO	R/W		FIFO: Data In/Data Out

Table 6 Hash/Encrypt Registers**Hash/Encrypt Control Register (HECNTL)**

This 16-bit Read/Write register, as shown in the table below, allows selecting configuration settings for the Hash/Encrypt subsystem.

Register Address (READ / WRITE)															
DSP				PCMCLA				PCI							
0x1000				0x0200 – 0x0201				0x0200 – 0x0201							

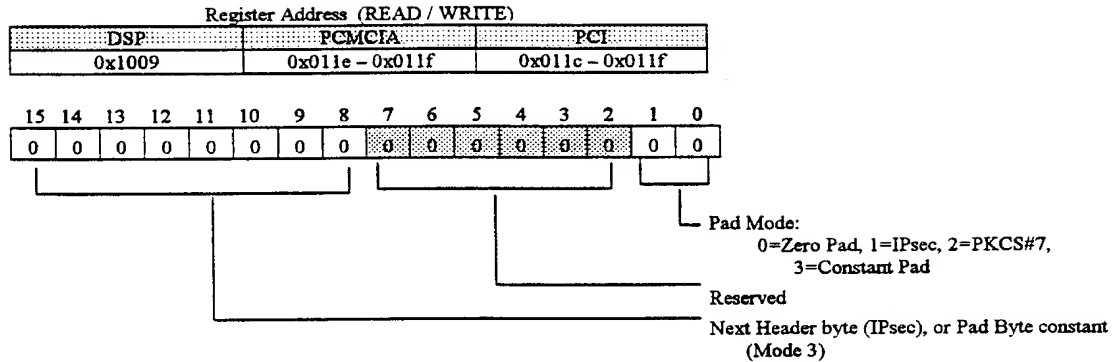
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Allow IV writes to context registers:
0 = allow software to write to IV
1 = ignore data written to IV registers

Reserved

Pad Control Register (HEPADCNTL)

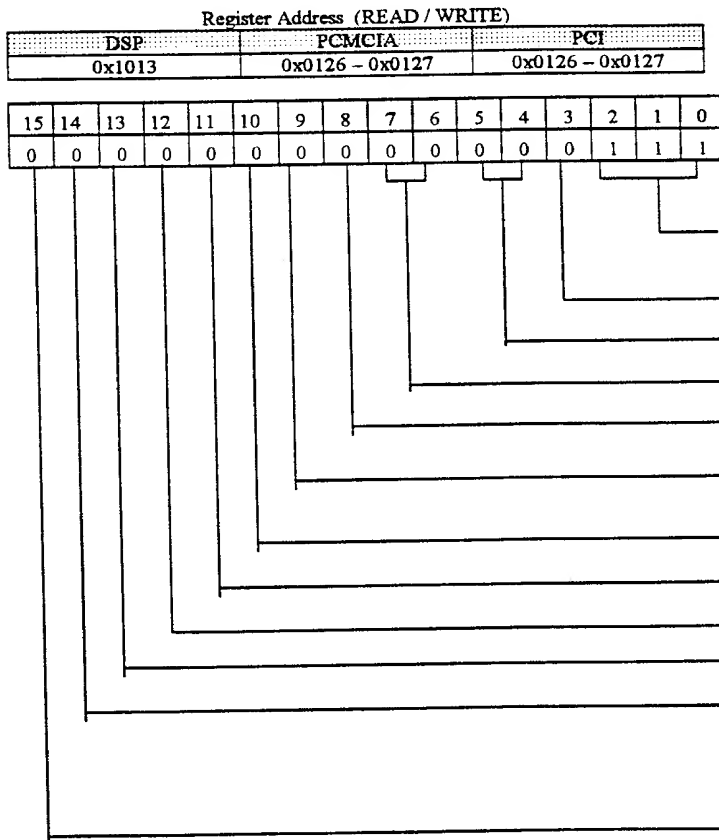
Shown in the table below are the bit definitions for the Pad Control Register:



Control Register (HECNTL)

Shown in the table below are the bit definitions for the Hash/Encrypt

Control Register:



Operation :

(0=encrypt; 1=decrypt; 2=hash; 3=hash-enc; 4=dec-hash; 5=hash-dec; 6=enc-hash; 7=idle)

1 = Triple-DES, 0 = DES

Mode: (0=ECB; 1=CBC; 2=OFB; 3=CFB)

feedback bits (0 = 64-bits; 1 = 8-bits; 2 = 1-bit)

1 = SHA, 0 = MD5

1 = initial hash state is algorithm constants
0 = use inner Digest as initial state

1 = load hash byte count, 0 = start from zero

1 = perform outer hash

1 = perform hash 'final' processing

1 = use context 1, 0 = use context 0

1 = offset is from start of 2nd sub-operation to start of 1st sub-operation.

0 = offset is from start of 1st sub-operation to start of 2nd sub-operation.

1 = decrypt key before use

Consume Pad Register (HECONSPAD)

This 16-bit Write-Only register, as shown in the table below, allows the Software to command a discard of the last 8-byte block of decrypted data in the Hash/Encrypt output FIFO. This is typically used to avoid the bus bandwidth of transferring the Pad block back to host memory.

Register Address (WRITE ONLY)															
DSP				PCMCIA				PCI							
0x1014				0x0228 – 0x0229				0x0228 – 0x0229							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x
<div style="border-top: 1px solid black; height: 10px; width: 100%;"></div> <div style="text-align: center;">Don't Care</div>															

Pad Status Register 0 (HEPADSTAT0)

Shown in the table below are the bit definitions for the Pad Status Register

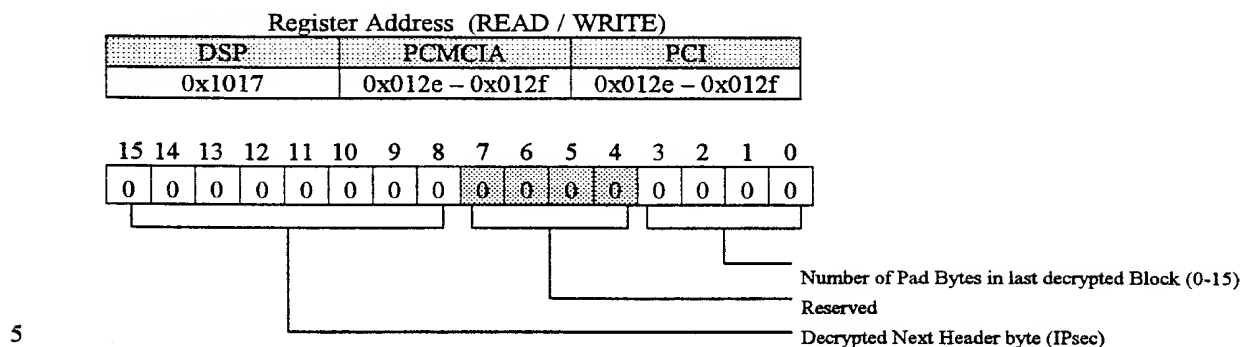
0:

Register Address (READ / WRITE)															
DSP				PCMCIA				PCI							
0x1016				0x012c – 0x012d				0x012c – 0x012d							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<div style="border-top: 1px solid black; height: 10px; width: 100%;"></div> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="border-top: 1px solid black; width: 30%;"></div> <div style="border-top: 1px solid black; width: 20%;"></div> <div style="border-top: 1px solid black; width: 20%;"></div> <div style="border-top: 1px solid black; width: 30%;"></div> </div> <div style="display: flex; justify-content: flex-end; margin-top: 5px;"> <div style="margin-right: 20px;">Number of Pad Bytes in last decrypted Block (0-15)</div> <div style="margin-right: 20px;">Reserved</div> <div>Decrypted Next Header byte (IPsec)</div> </div>															

Pad Status Register 1 (HEPADSTAT1)

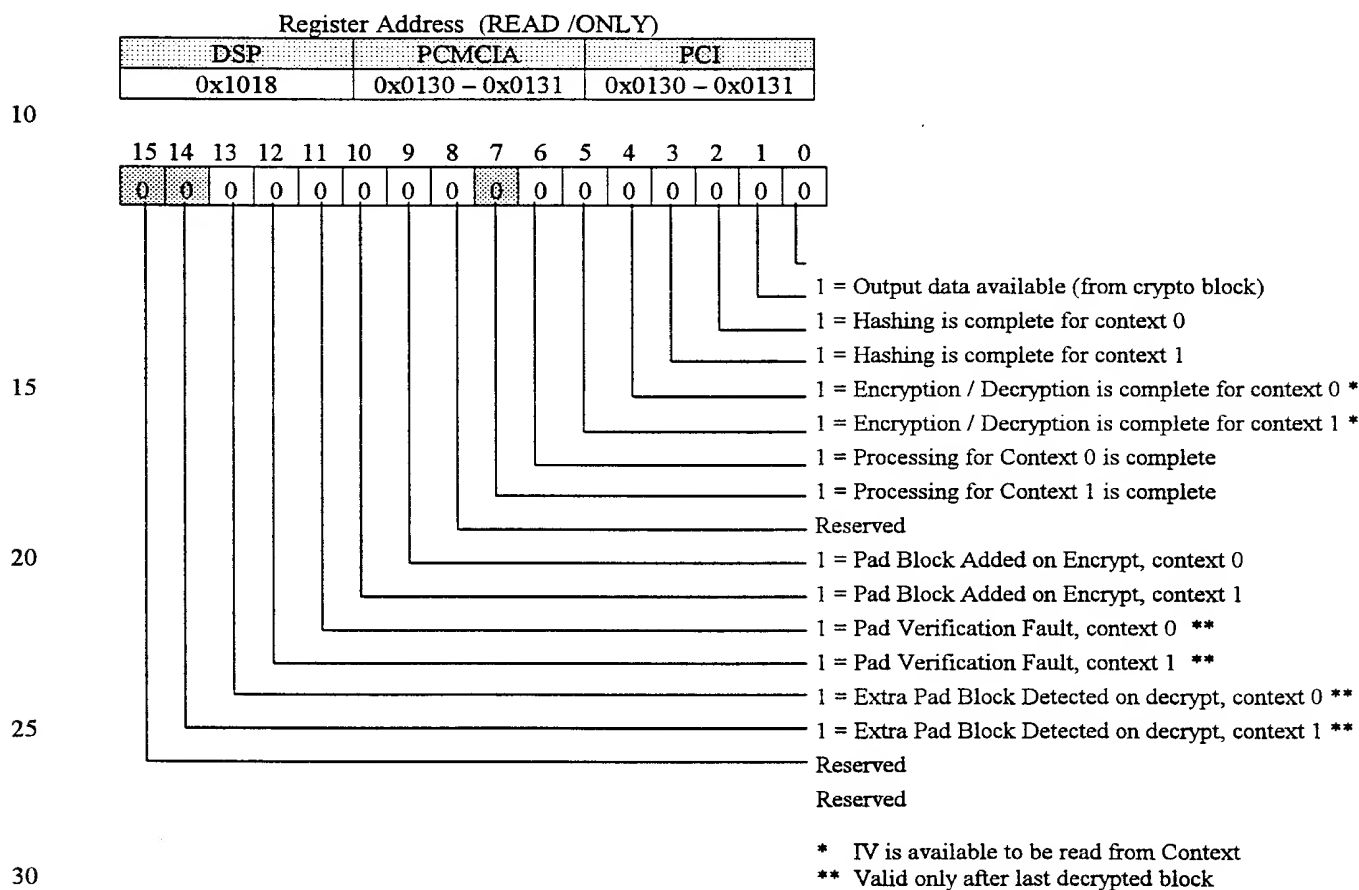
Shown in the table below are the bit definitions for the Pad Status Register

1:



General Status Register (HESTAT)

Shown in the table below are the bit definitions for the Hash/Encrypt General Status Register:



APPLICATION REGISTERS

The Application Registers (*AppRegs*) subsystem provides a bi-directional mailbox facility between a Host and the *CryptIC* and also contains some miscellaneous configuration registers. The mailbox is specifically designed to facilitate the transfer of CGX crypto commands back and forth between the *CryptIC* and the calling Host.

Overview

The *AppRegs* mailbox subsystem is designed to follow a 'ping-pong' protocol between *CryptIC* and Host. Default power-up ownership of the mailbox is given to the Host. After either side writes an entire message to the mailbox, it automatically switches state so that the other entity is given 'ownership' of the mailbox. Optionally, an Interrupt may be sent to the DSP/Host when the other side has finished writing a message.

Up to a 44-byte (22 word) message may be written to the mailbox by the DSP. The procedure should designate that the least-significant word of the message be written last, since that initiates an 'End-of-Message' sequence. (By using the least-significant word for termination, any length of message from 1 to 44 bytes can be supported.)

Application Registers Register Set

A set of memory-mapped control and status registers are provided in the 'Application Registers' subsystem. These are considered *Unprotected Registers*, and therefore are visible to either the DSP running in User mode or to an outside PCMCIA/PCI bus entity. They are summarized in Table 7 and described in detail in the following subsections.

(In the table below, 16-bit address refers to the DSP and 32-bit address refers to PCI/PCMCIA host.

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset efault	DESCRIPTION
APPLICATION REGISTERS					
0x1880- 0x1895	0x0000-2B	CGX Command	R/W		44-byte CGX command register
0x18A0	0x0040-41	Status	R	0x0000	Application registers status
0x18A1	0x0042-43	Lock	R/W		DSP/Host lock control
0x18A2	0x0044-47	Misc Status	R/W		Miscellaneous status bits: DSP <-> host
0x18A3	N/A	Select Delay	R/W		Delay configuration for memory pulse generation
0x18A4	N/A	Hash/Encrypt Byte Enable	R/W		Byte enables for data R/W to Hash/Encrypt block
0x18A5	N/A	Reset Violation Memtype/Addr	R	0x0000	Holds the memory type and Address of the last protection violation-induced Reset
0x18A6	0x004C-4F	Extmem Config	R/W		External memory configuration

Table 7 *Application Register Set*

Mailbox Data Register (MAILDAT)

5 This is actually a contiguous set of register comprising 44-bytes, as shown in the table below. The least significant word of this register has special properties. When written by the DSP, the lsword will cause a 'DSP_Wrote_Command' interrupt to be issued to the Host processor (if enabled). Similarly, a Host write will cause a 'Host_Wrote_Command' interrupt to be issued to the DSP processor (if enabled). In addition, writing the least-significant word will cause the ownership of the mailbox to automatically flip to the other party.

10

From the DSP perspective, the mailbox appears as 22 consecutive 16-bit locations:

DSP Register Address (READ / WRITE)

DSP Address	Mailbox Data
0x1895	Data Word 21 (msw)
0x1894	Data Word 20
0x1893	Data Word 19
.	.
.	.
.	.
0x1882	Data Word 02
0x1881	Data Word 01
0x1880	Data Word 00 (lsb) **

** End-of-Write Trigger Word

From the Host perspective, the mailbox appears as 11 consecutive 32-bit locations which are byte-addressable, as shown in the table below:

5

Host Register Address (READ / WRITE)

Host Address	Mailbox Data
0x002B	Data Byte 43 (msb)
0x002A	Data Byte 42
0x0029	Data Byte 41
.	.
.	.
.	.
0x0002	Data Byte 02
0x0001	Data Byte 01
0x0000	Data Byte 00 (lsb) **

** End-of-Write Trigger Byte

DSP AppRegs Status Register (DSPAPPSTAT)

This 16-bit Read-only register, as shown in the table below, allows the DSP Software to monitor the status of the Mailbox.

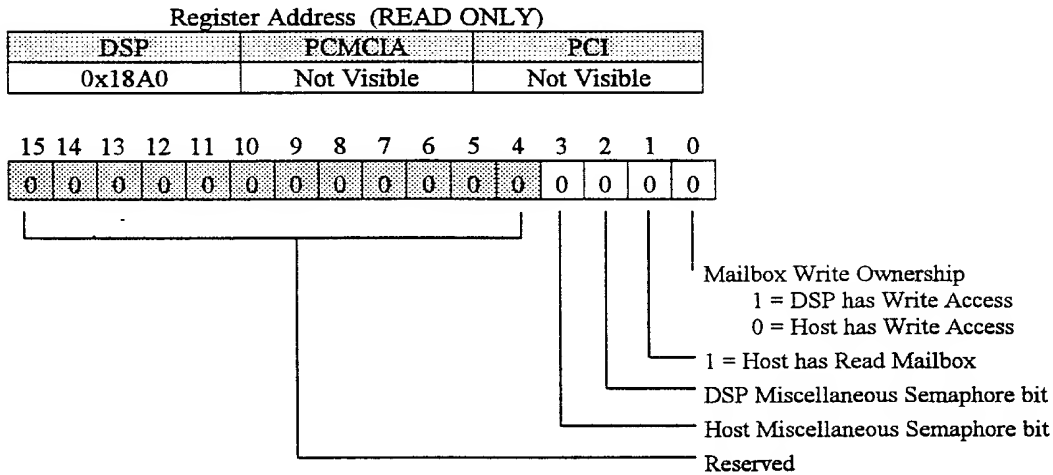
10 Bit #0, Mailbox Write Own, allows the DSP to determine who may next write into the Mailbox.

Bit #1, Host has read Mailbox, allows the DSP to monitor when the Host processor has received a transmitted message in the Mailbox. Bit 1 will be latched as a '1' when the Host has read the least significant data byte of the Mailbox. As soon as the DSP

reads DSPAPPSTAT, this bit will automatically be cleared and re-armed for the next Host Mailbox read.

Bits #2 & 3 are read-only status bits which reflect the semaphores written (and read) in the DSPSEMA and HOSTSEMA registers.

5



10

15

Host AppRegs Status Register (HOSTAPPSTAT)

This Read-only register, as shown in the table below, allows the Host Software to monitor the status of the Mailbox.

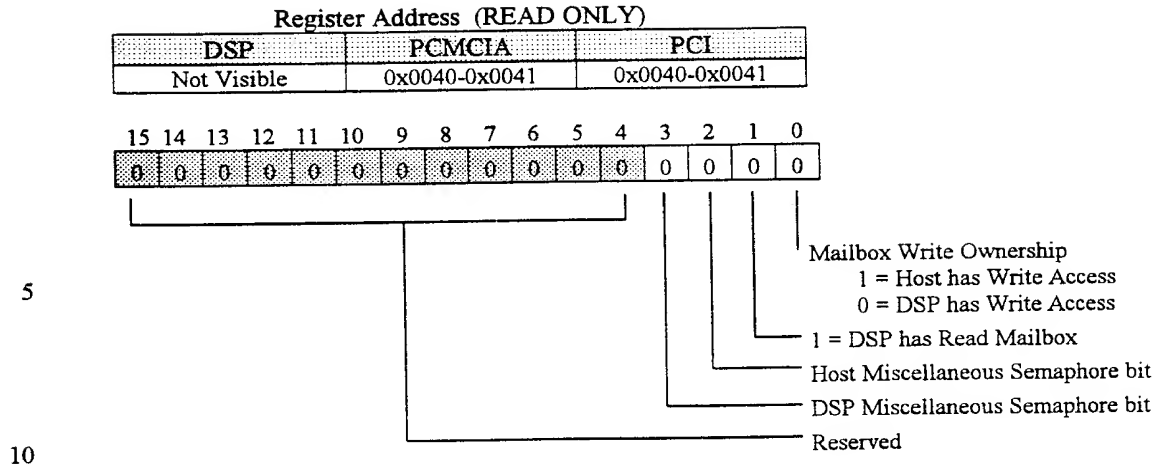
20

Bit #0, Mailbox Write Own, allows the Host to determine who may next write into the Mailbox.

25

Bit #1, DSP has read Mailbox, allows the Host to monitor when the DSP processor has received a transmitted message in the Mailbox. Bit 1 will be latched as a '1' when the DSP has read the least significant data byte of the Mailbox. As soon as the Host reads HOSTAPPSTAT, this bit will automatically be cleared and re-armed for the next DSP Mailbox read.

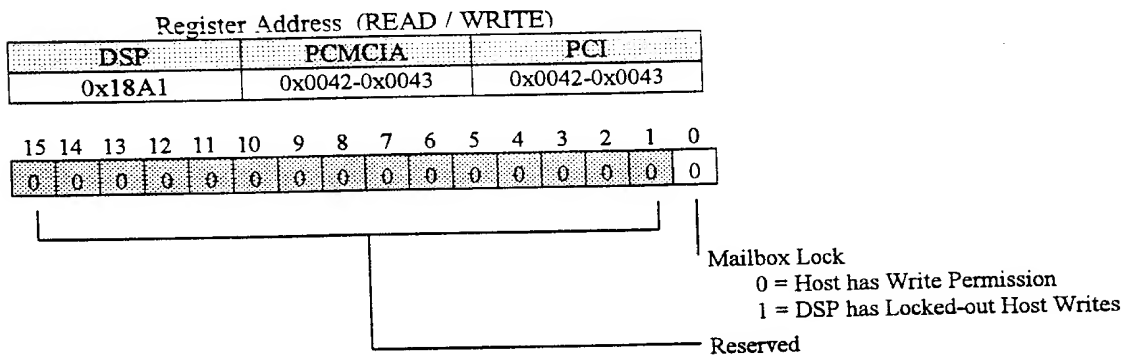
Bits #2 & 3 are read-only status bits which reflect the semaphores written (and read) in the DSPSEMA and HOSTSEMA registers.



AppRegs Lock (APPLOCK)

15
 This 16-bit Read/Write register, as shown in the table below, allows the DSP to lock-out Host write access to the Command/Response register. Setting this register to 0x01 effectively makes the Mailbox one-way from the DSP to the Host.

From the Host perspective, this register is read-only.



DSP (Host) Miscellaneous Semaphore Register (DSPSEMA, HOSTSEMA)

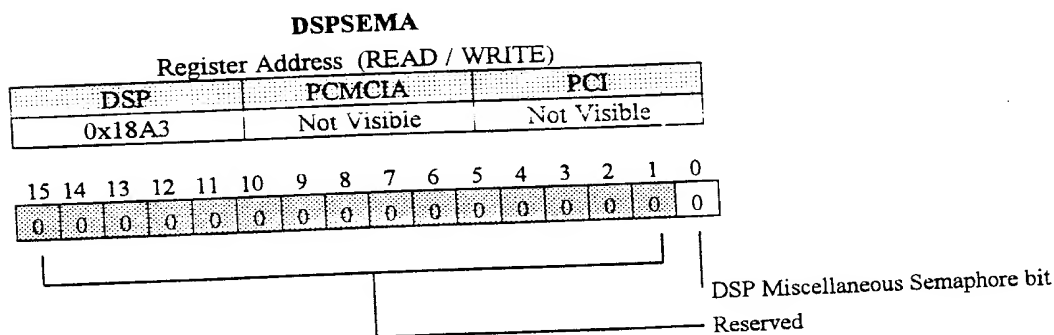
Viewed from the DSP, this 16-bit Read/Write register, as shown in the table below, allows the DSP Software to assert a semaphore bit to be read by the Host via the AppRegsStatus register.

5 When viewed from the Host, the lsb of this 32-bit Read/Write register allows the Host Software to assert a semaphore bit to be read by the DSP via the AppRegsStatus register.

The CGX Kernel may be programmed to use these two bits for ownership arbitration of the Hash/Encrypt subsystem. In order to enable this feature, the appropriate CIS initialization bit must be set when calling the CGX_INIT routine. 10 Refer to the CGX Software Users Guide for more details on the behavior of this function.

If the above CGX usage of these semaphores is not selected, then they may be used for any general purpose DSP to Host signalling.

15



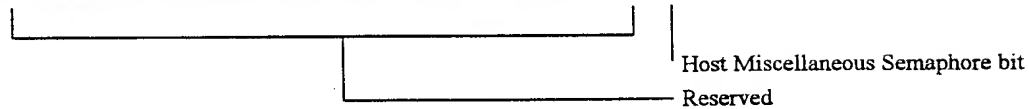
20

HOSTSEMA

Register Address (READ / WRITE)

DSP	PCMCIA	PCI
Not Visible	0x0044-0x0047	0x0044-0x0047

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



5

Hash/Encrypt Byte Enable Register (HEBYTEEN)

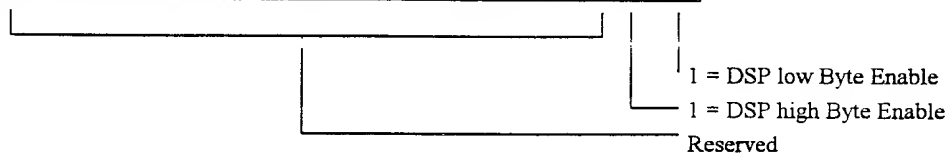
This 16-bit Read/Write register, as shown in the table below, allows the DSP Software to manage reading and writing fractional words in or out of the Hash/Encrypt FIFO's. Since the DSP is oriented towards a 16-bit bus width, but the FIFO's can manipulate data down to byte granularity, it is necessary for the DSP to specify whether its low or high order byte is to be transferred in or out of the FIFOs.

10

Register Address (READ ONLY)

DSP	PCMCIA	PCI
0x18A0	Not Visible	Not Visible

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



15

Reset Reason/Instruction Register (RSTREASON)

This 16-bit Read/Write register, as shown in the table below, allows the DSP processor to determine the offending address and memory type which caused a Kernel protection violation reset. The contents of this register are preserved across a *CryptIC* reset (although of course, not across a power-cycle.) This is useful in de-bugging

20

software which is written for the User Mode of the *CryptIC*. Refer to section previous described on Kernel Mode Control for more information about the protection features of the *CryptIC*.

The four memory types are as follows:

PM Instruction Fetch	An instruction fetch was made either to one of the four blocks of Kernel ROM (i.e. the 4 lsb's of PMOVLAY were set to 0xC, D, E or F and a fetch was made between address 0x2000 and 0x3FFF), or to an internal program memory (PM) location which has been locked-in as <i>Protected Memory</i> by an <i>Extended Mode</i> program.
DM Data Fetch	A data fetch was made to an internal data memory (DM) location which has been locked-in as <i>Protected Memory</i> by the Kernel. With the standard CGX kernel and no <i>Extended Mode</i> programs active, none of the internal DM should be protected.
PM Data Fetch	A data fetch was made to an internal program memory (PM) location which has been locked-in as <i>Protected Memory</i> by the Kernel. With the standard CGX kernel and no <i>Extended Mode</i> programs active, none of the internal PM should be protected.
Protected Registers/ Kernel RAM Data Fetch	A data fetch was made to either the Kernel RAM area or one of the Protected crypto registers. This would mean that the 4 lsb's of DMOVLAY were set to 0xF and a data fetch was made between address 0x0000 and 0x17FF

5 Due to a design 'feature' of the *CryptIC*, this register must be read twice, ignoring the first data read.

Register Address (READ / WRITE)		
DSP	PCMCLA	PCI
0x18A5	Not Visible	Not Visible

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

10

14-bit Memory Address which caused violation

Memory Type which caused violation

15

00 = PM Instruction Fetch

01 = DM Data Fetch

10 = PM Data Fetch

11 = Protected Registers/Kernel RAM data fetch

External Memory Configuration Register (EXMEMCFG)

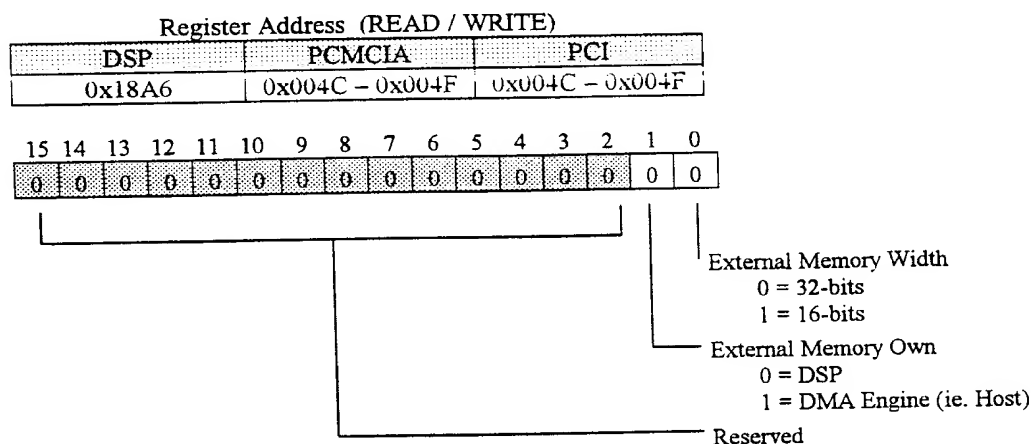
This 16-bit Read/Write register, as shown in the table below, allows selecting configuration settings for the external memory subsystem.

Bit 0 configures the nature of the external memory bus. It should normally be programmed at *CryptIC* power-up and left unchanged. One effect of this bit is that it changes the behavior of external address bit [0] and the CS32 and DMS pins. In 16-bit memory mode, only DMS should be used and A0 functions to select even and odd words of memory.

In 32-bit mode, then memory devices must be used which allow 16-bit bus enables (e.g. either a 32-bit wide device with upper and lower 16-bit enables, or a pair of 16-bit wide devices). In this case, A0 should be left disconnected. DMS will select the 'lower' 16-bits of external memory, and CS32 will select the 'upper' 16-bits.

Bit #1, External Memory Own, allows the DSP to arbitrate who owns the external memory bus: either the DSP or the DMA controller (and therefore also the Host processor). Typically, software will leave this bit set to 1 except when external memory must be accessed by the DSP. This allows PCI Target mode transfers to occur to ExtMem (since a Target transfer cannot always be predicted by the DSP).

IRE's CGX software always leaves this bit set to 1 (DMA controller owns), unless it is in the middle of an operation which requires DSP access to ExtMem.



INTERRUPT CONTROLLER

Interrupt Controller Overview

5 The *CryptIC* enhances the existing Interrupt controller within the ADSP 2183 DSP with some additional functions related to the Crypto functional blocks and the external Host bus interfaces. Two additional Interrupt Controller subsystems have been added to the basic 2183 Interrupt Controller as shown in Figure 10.

10 The DSP Interrupt Controller allows programming between 1 and 7 sources for the IRQ2 interrupt to the DSP. The DIMASK register provides the Mask to select which interrupt source is enabled. A pair of status registers, DUSTAT and DMSTAT, allow the DSP firmware to read the status of any interrupt source either before or after the Mask is applied.

15 The Host Interrupt Controller allows programming between 1 and 4 sources for the HostIntO interrupt output signal (which may be connected to the interrupt input of the Host system). The HIMASK register provides the mask to select which interrupt source is enabled. A pair of status registers, HUSTAT and HMSTAT, allow the Host firmware to read the status of any interrupt source either before or after the Mask is applied.

Interrupt Source Descriptions

20 The DSP and Host Interrupt Controllers support the following additional interrupt sources, as shown in the table below.

Name	Description
Host Interrupt	This is a software-forced interrupt towards the DSP from the Host.
DSP Interrupt	This is a software-forced interrupt towards the Host from the DSP.
H/E Context 0 Done	This indicates that the Hash/Encrypt engine within the <i>CryptIC</i> has just finished fully processing a command for Context 0. It will occur after either the Length bytecount has decremented to 0 or if the H/E Control word is written with an idle command (111 in the 3 lsb's).
H/E Context 1 Done	Same as above, but for Context 1.
Host wrote Command	The Host Processor has just written the least-significant word of the Application Registers mailbox. This notifies the DSP of this event.
DSP wrote Command	The DSP has just written the least-significant word of the Application Registers mailbox. This notifies the Host of this event.
Master DMA Transfer Done	Indicates that a DSP-initiated Master DMA transaction is complete. This is triggered from bit 3 of the DMA Status/Config. Register going from 1 to 0.
Master DMA Transfer Queued	Indicates that a DSP-initiated Master DMA transaction has just been queued in the double-buffered holding register, thus there is no room for another DSP DMA command. This is triggered from bit 15 of the DMA Status/Config. register going from 0 to 1.
Host External Memory Conflict	A DMA transfer was attempted to or from External Memory, but the ownership bit was not set to allow the transfer (ie. the DSP had ownership.) See EXMEMCFG register in section 0.
Hash/Encrypt Error	A Hash/Encrypt error occurred. The error code is given in the H/E Error Code register.
IRQ2	This is simply the IRQ2 external pin on the <i>CryptIC</i> . This allows the DSP to 'OR' it in with the other crypto-related interrupt sources

Table 8 *Interrupt Sources***Interrupt Control Registers (INTC)**

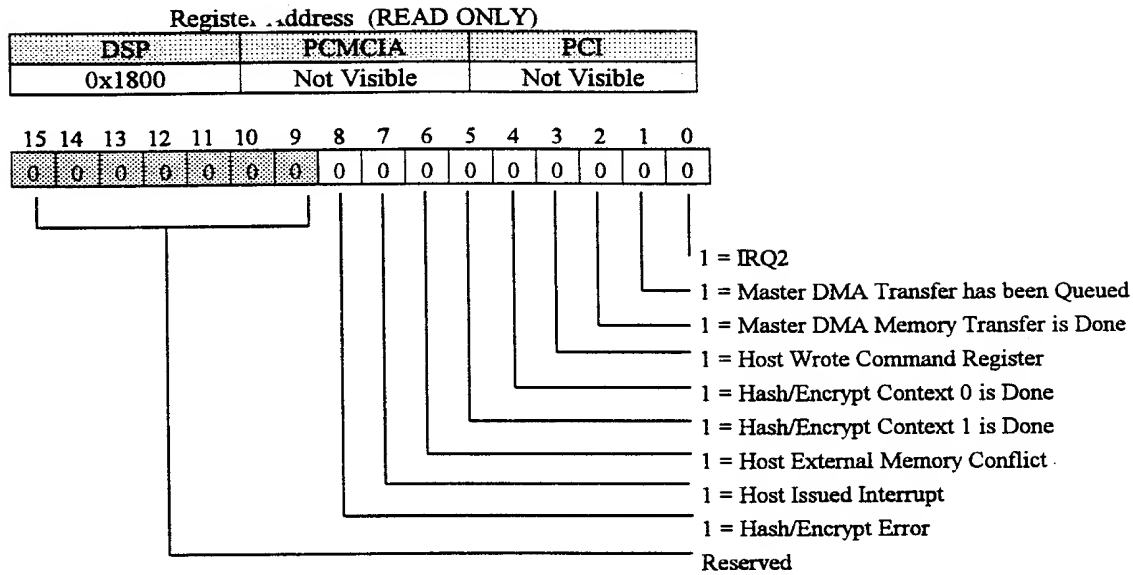
5 The Interrupt Control and Status Registers, as shown in the table below, consist of two sets of 6 registers which allow the DSP or the Host system to enable or disable crypto interrupts, check the status of the most recent interrupt, force an interrupt, etc.

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset efault	DESCRIPTION
INTERRUPT CONTROLLER REGISTERS					
DSP-Visible Registers:					
0x1800	N/A	DSP Unmasked Status	R		Interrupt source current states – Prior to mask
0x1801	N/A	DSP Masked Status	R		Interrupt source current states – Post mask
0x1801	N/A	DSP Clear Int	W		Clear selected Interrupt
0x1802	N/A	DSP Mask Control	R/W		Interrupt mask register
0x1803	N/A	DSP Int Config.	R/W		DSP Interrupt configuration register
0x1804	N/A	Force Host Int	W		Force interrupt to Host (PCI/PCMCIA)
0x1805	N/A	H/E Error Code	R		Provides the H/E Error Code
Host-Visible Registers:					
N/A	0x0080-0081	Host Unmasked Status	R		Interrupt source current states – Prior to mask
N/A	0x0084-0085	Host Masked Status	R		Interrupt source current states – Post mask
N/A	0x0084-0085	Host Clear Int	W		Clear selected interrupt
N/A	0x0088-0089	Host Mask Control	R/W		Interrupt mask register
N/A	0x008C-008D	Host Int Config.	R/W		Host interrupt configuration register
N/A	0x0090-0091	Force DSP Int	W		Force interrupt to DSP
N/A	0x0092-0093	H/E Error Code	R		Provides the H/E Error Code

Table 9 Interrupt Register Set

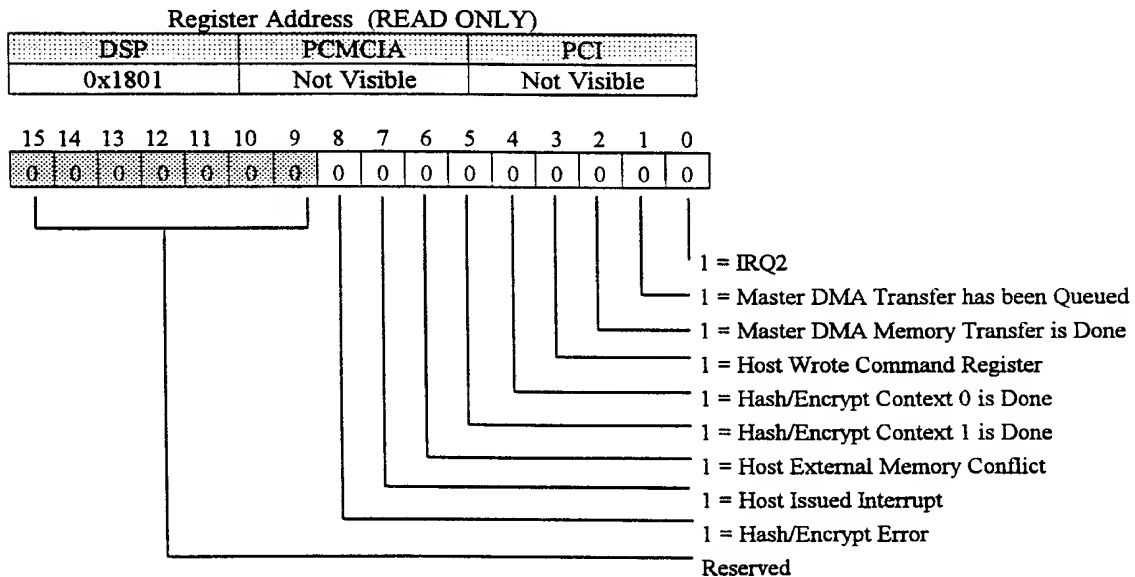
DSP Unmasked Status Register (DUSTAT)

5 This 16-bit Read-Only register, as shown in the table below, provides interrupt status visibility to the DSP, prior to the interrupt mask being applied. Thus, the DSP can view all potential sources of incoming interrupt. All of these sources, whether masked in or out, will be latched in this register and must be cleared using the DICLR register in order to view a subsequent interrupt.



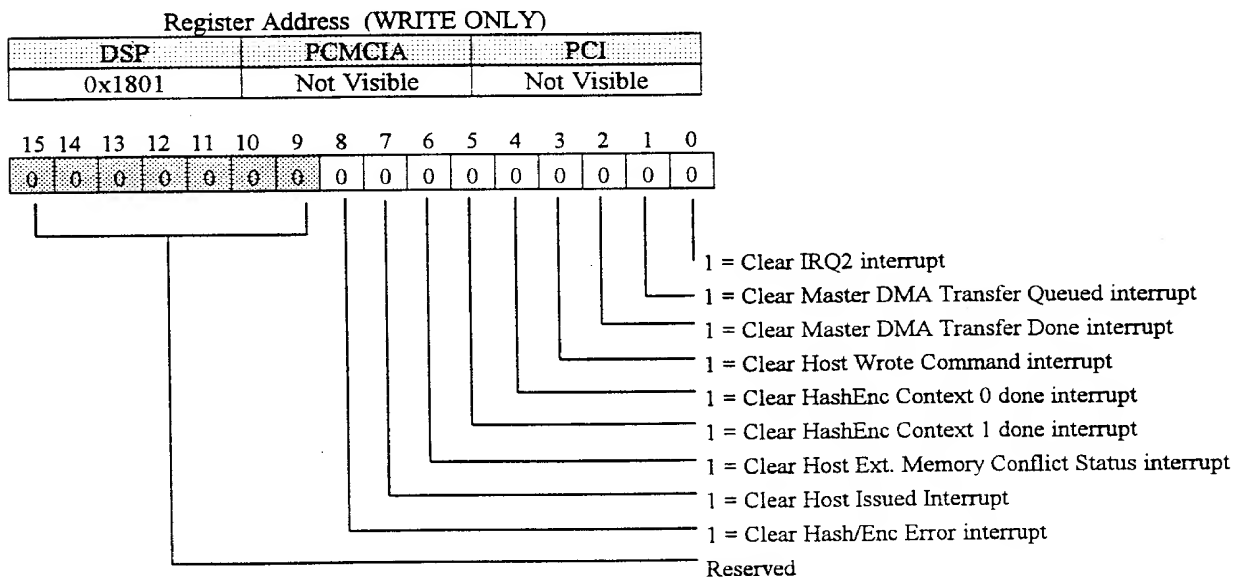
DSP Masked Status Register (DMSTAT)

This 16-bit Read-Only register, as shown in the table below, provides Interrupt Status visibility to the DSP, after the Interrupt Mask is applied. This lets the DSP view the selected sources of Interrupts which are directed to IRQ2. (Note that the DSP must enable IRQ2 via the IMASK register and the global ENA INTS command must be issued in order to actually receive the interrupt.) As with the Unmasked status register, all interrupt bits are latched and must be cleared using the DSP Clear Interrupt register.



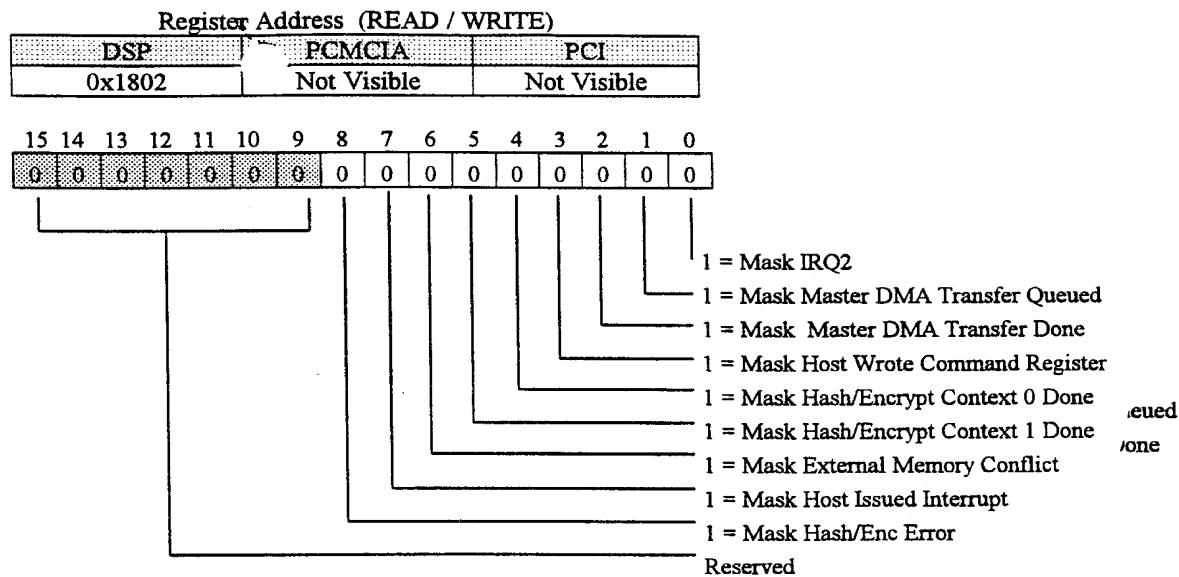
DSP Clear Interrupt Register (DICLR)

This 16-bit Write-Only register allows the Processor to Clear pending Interrupts. It is located at the same address as the Masked Status Register (Write vs. Read) which facilitates performing a read to detect pending interrupts followed by a write of the same bits in order to clear the latched interrupt status.



DSP Mask Control Register (DIMASK)

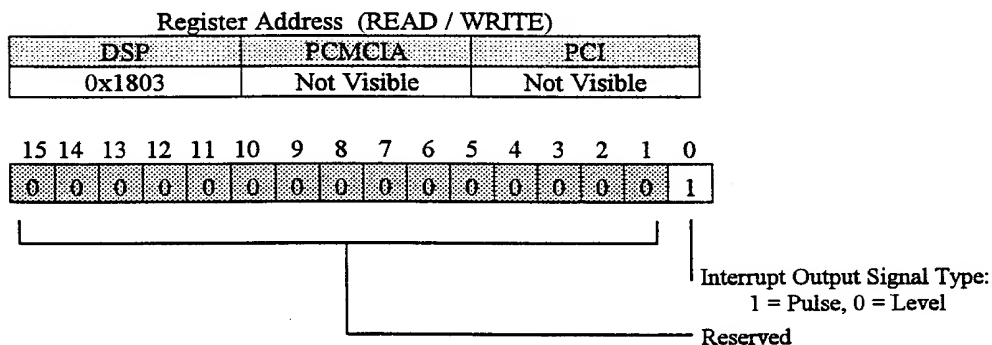
This 16-bit Read/Write register, as shown in the table below, allows configuring the Interrupt Masks for the Crypto subsystem.



DSP Interrupt Configuration Register (DICFG)

This 16-bit Read/Write register, as shown in the table below, allows configuring the Interrupt type which will be fed into the IRQ2 interrupt line of the *CryptIC*'s DSP. Note that this only effects the final output of the Interrupt subsystem.

Configuring for *Pulse* will cause the internal interrupt signal to pulse low for two clock cycles when activated. When set for *Level*, the interrupt signal will be set low until cleared by the DSP (i.e. it will follow the bit in the Masked Status Register).



Force DSP Interrupt Register (DIFRC)

This 16-bit Write-Only register, as shown in the table below, allows the DSP Software to Force a Crypto subsystem Interrupt to the external Host

processor. The data contents of the Write operation are ignored – any Write to this address will cause an interrupt, if enabled by the HIMASK register.

Register Address (WRITE ONLY)															
DSP				PCMCIA				PCI							
0x1804				Not Visible				Not Visible							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X

|

Don't Care

DSP or Host H/E Errors Register (DSPHERR, HOSTHERR)

This 32-bit Read/Write register, as shown in the table below, provides the error code which resulted in a H/E Error interrupt. Reading a '1' in a bit position within this register indicates the type of error which occurred. In order to clear the error latch, 0's should be written to the desired bit positions.

Register Address (READ/WRITE)

DSP	PCMCIA	PCI
0x1805	Not Visible	Not Visible

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 = Write attempted to Hash FIFO, buff not rdy
 1 = Write attempted to Crypt FIFO, buff not rdy
 1 = Attempted to use Red key when not allowed
 1 = Attempt to write Control word when not rdy
 1 = Overflow operation on Length count (more bytes written than non-0 Length specifies)
 Reserved

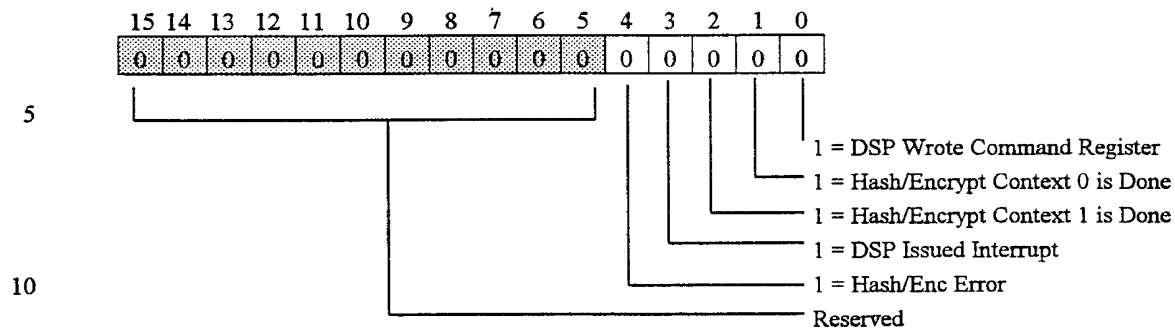
Host Unmasked Status Register (HUSTAT)

This 32-bit Read-Only register, as shown in the table below, provides interrupt status visibility to the Host, prior to the interrupt mask being applied. Thus, the Host can view all potential sources of incoming interrupt. All of these sources, whether masked in or out, will be latched in this register and must be cleared using the HICLR register in order to view a subsequent interrupt.

can view all potential sources of incoming interrupt. All of these sources, whether masked in or out, will be latched in this register and must be cleared using the HICLR register in order to view a subsequent interrupt.

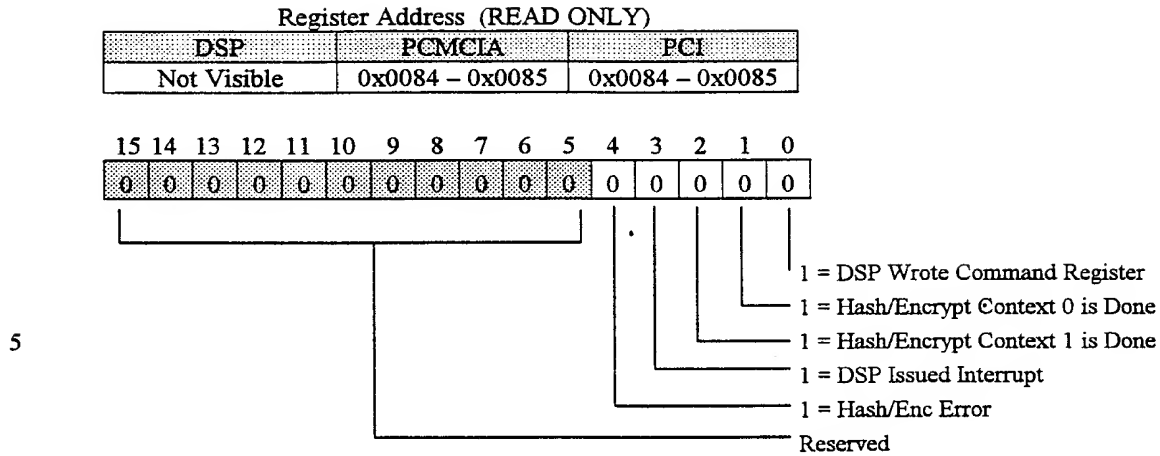
Register Address (READ ONLY)

DSP	PCMCIA	PCI
Not Visible	0x0080 – 0x0081	0x0080 – 0x0081



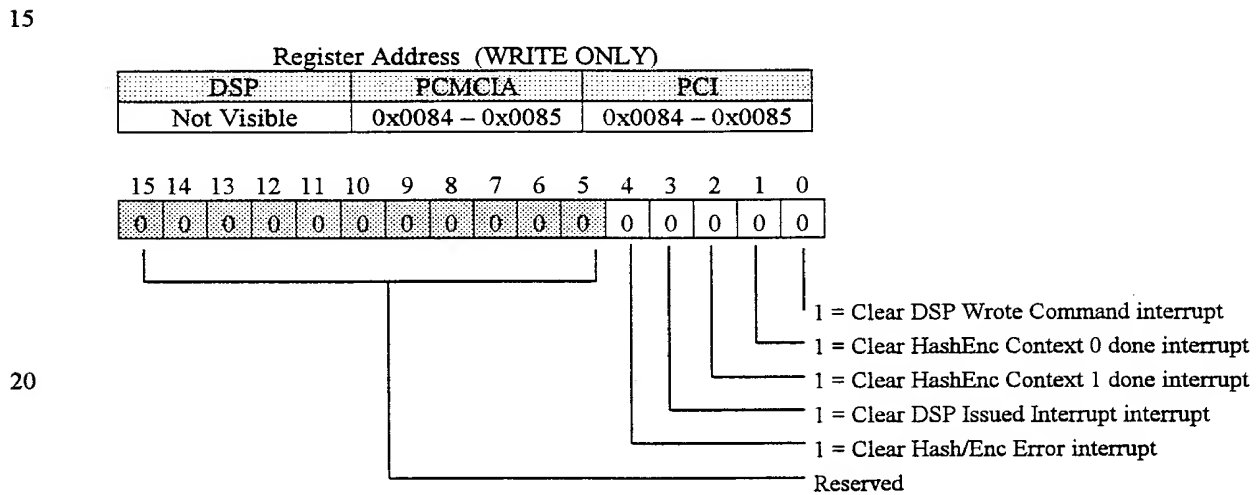
Host Masked Status Register (HMSTAT)

This 32-bit Read-Only register, as shown in the table below, provides Interrupt Status visibility to the Host, after the Interrupt Mask is applied. This lets the Host view the selected sources of Interrupts which are directed to PF7 which is normally connected to the Host bus interrupt. As with the Unmasked status register, all interrupt bits are latched and must be cleared using the Host Clear Interrupt register.



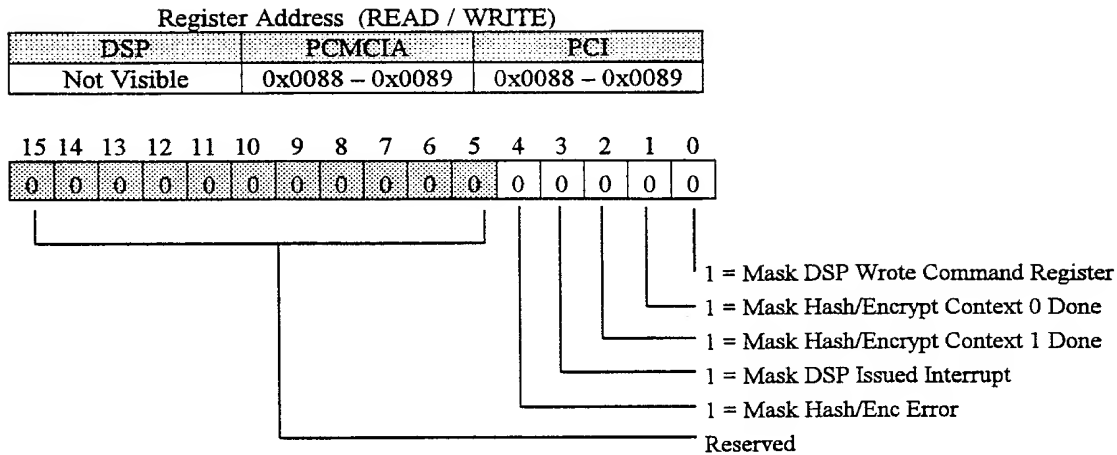
Host Clear Interrupt Register (HICLR)

10 This 32-bit Write-Only register, as shown in the table below, allows the Host to Clear pending Interrupts. It is located at the same address as the Masked Status Register (Write vs. Read) which facilitates performing a read to detect pending interrupts followed by a write of the same bits in order to clear the latched interrupt status.



Host Mask Control Register (HIMASK)

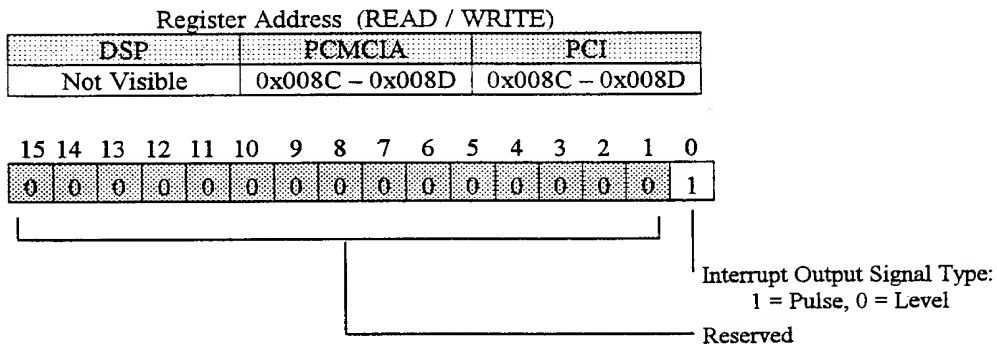
25 This 32-bit Read/Write register, as shown in the table below, allows configuring the Interrupt Masks for the Crypto subsystem.



Host Interrupt Configuration Register (HICFG)

This 32-bit Read/Write register, as shown in the table below, allows configuring the Interrupt type which will be fed to the PF7 interrupt line out of the *CryptIC*. Note that this only effects the final output of the Interrupt subsystem.

Configuring for *Pulse* will cause the PF7 interrupt output to pulse low for two clock cycles when activated. When set for *Level*, the interrupt signal will be set low until cleared by the Host (i.e. it will follow the bit in the Masked Status Register).



Force Host Interrupt Register (HIFRC)

This 32-bit Write-Only register, as shown in the table below, allows the Host Software to Force a Crypto subsystem Interrupt to the DSP processor. The data

contents of the Write operation are ignored – any Write to this address will cause an interrupt, if enabled by the DIMASK register.

Register Address (WRITE ONLY)		
DSP	PCMCIA	PCI
Not Visible	0x0090 – 0x0091	0x0090 – 0x0091

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Don't Care

5

SERIAL EEPROM

Serial EEPROM Overview

The *CryptIC* provides an interface port for connection of a 2kbit (256-byte) serial EEPROM (93C66 or equiv.). The principal purpose of this is to allow automatic boot-loading of the PCI or PCMCIA port configuration parameters. The serial EEPROM is not mandatory however, since the DSP may directly program these configuration settings into the PCI/PCMCIA bus core using the registers described below. It should be noted however, that the DSP must be able to program these settings before it can communicate using the PCI/PCMCIA bus. This implies that the *CryptIC* could not be boot-loaded from the PCI/PCMCIA bus unless a EEPROM was connected.

A secondary use of this EEPROM port can be application-specific. Since there is extra unused space in the EEPROM, a user application may use the remaining space for non-volatile storage of a relatively small amount of data. Examples could include: A Black KEK, a digital certificate, a user authentication 'master' password, etc.. Since the EEPROM is directly connected to the *CryptIC* and can only be controlled by the DSP, there is a certain amount of intrinsic protection to this data store.

For PCI applications, the PCI configuration parameters occupy the first 13 locations of the EEPROM, leaving 243 bytes remaining for user-specific applications. In Cardbus systems, there is additional CIS storage required after the first 13 locations.

For PCMCIA applications, the CIS configuration parameters occupy the first TBD locations of the EEPROM, leaving TBD bytes remaining for user-specific applications.

5 For applications which do not use either the PCI or the PCMCIA buses, the entire EEPROM (256 bytes) is available for applications.

In order for user applications to utilize the serial EEPROM, a low-level software driver must be written on the DSP to implement the unique bit-level protocol of the serial EEPROM. The EEPROM Command/Status port simply provides read/write access directly to the four EEPROM signal lines: Serial Clock (SK), Data In
10 (DI), Data Out (DO), and Chip Select (CS). Refer to the manufacturers datasheet for the EEPROM for information on its protocol.

'Manual' Host Bus Configuration

As mentioned above, the DSP can override the automatic bootloading of the
15 Host bus (PCI/PCMCIA) static configuration parameters from a EEPROM. The sequence of operations is as follows:

- a) Set [bit 1] of the EECMDSTAT register to '1' to stop the autoloading state machine.
- b) Write valid configuration into the first 13 EEPROM registers.
- 20 c) Set [bit 0 & bit 1] of the EECMDSTAT register to '1' to remove the EEPROM busy indication and enable the Host bus.
- d) Poll [bit 2] of the EECMDSTAT register to verify a '0' (not busy).

EEPROM Control Registers

25 The EEPROM Control and Status Registers, as shown in the table below, consist of 14 registers which allow the DSP to directly configure the PCI or PCMCIA

configuration registers. The last register, the Command/Status register allows general control of the serial EEPROM interface.

ADDRESS (16 BIT)	ADDRESS (32 BIT)	REGISTER NAME	R/W	Reset Default	DESCRIPTION
SERIAL EEPROM REGISTERS					
0x1900	N/A	Device ID	R/W		16-bit PCI device ID
0x1901	N/A	Vendor ID	R/W		16-bit PCI vendor ID (11D4h)
0x1902	N/A	Rev ID/Class	R/W		8-bit chip Revision ID, 8-msb's of PCI Class Code
0x1903	N/A	Class Code	R/W		remaining 16-lsb's of PCI Class Code
0x1904	N/A	Header Type/Int	R/W		PCI header type & Interrupt Pin
0x1905	N/A	Subsystem ID	R/W		16-bit PCI Subsystem ID
0x1906	N/A	Subsystem Vendor ID	R/W		16-bit Subsystem Vendor ID
0x1907	N/A	Max Lat. Min Gnt	R/W		Maximum Latency. Min Grant parameters
0x1908	N/A	Cardbus1	R/W		lower 16-bits of Cardbus CIS pointer
0x1909	N/A	Cardbus2	R/W		upper 16-bits of Cardbus CIS pointer
0x190A	N/A	Baddr mask1	R/W		Specifies 1 = modifiable 0 = our addresses
0x190B	N/A	Baddr mask2	R/W		Upper 16 bits
0x190C	N/A	CIS Size	R/W		CIS Size spec 16-bit (Upper 8 bits are 0)
0x190F	N/A	Cmd/Status	R/W		EEPROM Command and Status Register

EEPROM Command/Status Register (EECMDSTAT)

5 This 16-bit Read/Write register, as shown in the table below, allows control and status monitoring of various serial EEPROM functions. It also provides access to the input/output lines of the EEPROM, if enabled.

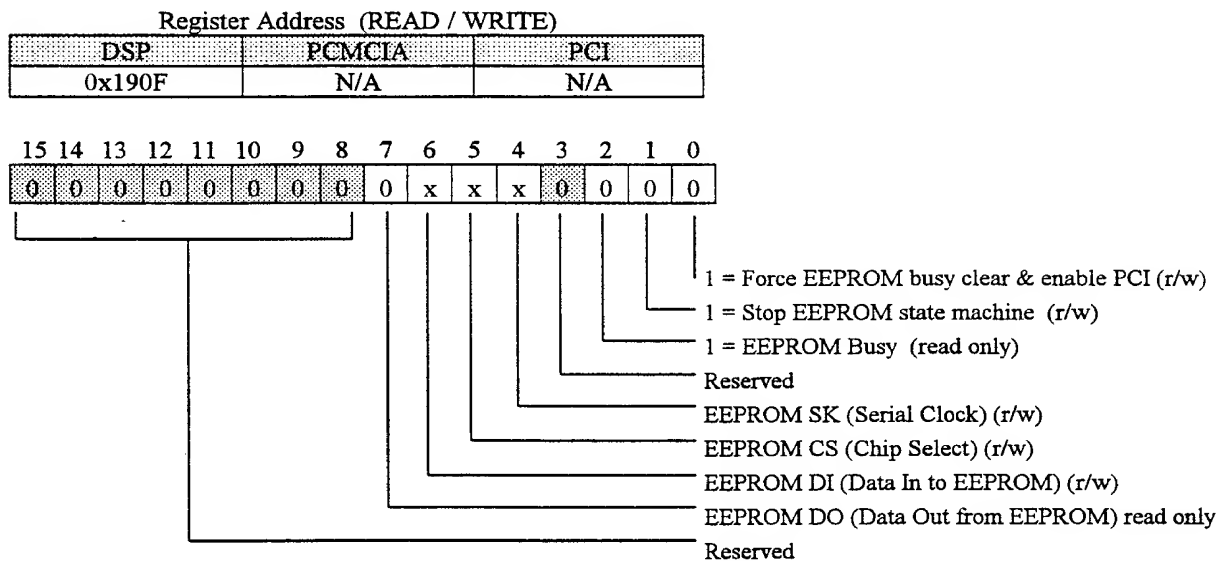
Bit 0: Clear EEPROM busy indication. The DSP should write a 1 to this bit after it has loaded the 13 PCI configuration registers.

10 Bit 1: Allows the DSP to shut-off the automatic hardware state machine which attempts to automatically download the PCI config/PCMCIA CIS into the bus interface core. It must be set to a 1 in order to program the first 13 EEPROM registers.

15 Bit 2: This bit indicates whether the hardware state machine is still attempting to load the configuration from the EEPROM into the bus core. It must be a 0 to enable

the PCI interface. It will clear to 0 if state machine finishes or if the DSP writes a 1 to bit 0.

Bits [7:4] allow the DSP direct access to the corresponding pins on the EEPROM device. For these register bits to be active, a '1' must be written to the *Stop EEPROM State Machine* [bit 1] of this register.



BOOT LOADING

Boot Loading Overview

The *CryptIC* provides multiple modes of boot-loading its operating software. Virtually all implementations of the *CryptIC* require at least some application-specific code running on its internal DSP. Of course, the CGX security kernel is hard-coded into ROM on the device and does not need to be boot-loaded from the outside.

The boot modes are as follows:

- Byte Memory Booting (BDMA)
- Host Bus Booting
 - From PCI
 - From PCMCIA

- From IDMA

- Local ROM Booting

Byte Memory Booting

5 In order to bootload the *CryptIC* from Byte memory, an 8-bit wide PROM must be connected to the *CryptIC*'s external memory bus in the 'Byte Memory' space (using BMS chip select).

To specify bootloading from Byte memory via the BDMA port, the following mode pin states must be set:

10 MMAP = 0
 BMODE = 0
 BUS_MODE = Don't care
 BUS_SEL = Don't care

15 When the Crypt powers-up, it will automatically begin DMA'ing the first 32 Program Memory words (96 bytes) into the base of internal Program Memory (PM). It will then begin program execution at address 0x0000 with a BDMA interrupt pending.

20 In fact, this bootloading process is identical to that for a standard ADSP 2181/2183 device. Refer to the Analog Devices ADSP-2100 Family User's Manual for more information on 'Boot Memory Interface'.

Host Bus Booting

The *CryptIC* may also be bootloaded via the Host processor bus interface. The procedure is slightly different, depending on which bus mode is selected.

PCI Bus Booting

In order to bootload from the PCI bus, the following mode pin states must be set:

MMAP = 0

5 BMODE = 1

BUS_MODE = 1

BUS_SEL = 0

In addition, a serial EEPROM must be connected to the *CryptIC* in order for it to automatically load its PCI default configuration parameters and enter the PCI-
10 Enabled state.

Once the *CryptIC* is powered-up, its PCI core will automatically initialize itself and the DSP will enter a Reset state. It will then wait for code to be bootloaded from the PCI host.

The PCI host will initially program the necessary data in the PCI Configuration
15 registers of the *CryptIC* (ie. Base Address Registers, etc.), and then may begin downloading a bootloader code segment into the *CryptIC*'s internal PM memory space. Downloading occurs by using Target mode IDMA writes into the internal PM space. See section PCI Address Map described previously for additional information.

The procedure dictates that PM address 0x0000 be loaded last, since this
20 causes program execution to begin automatically (Reset is deasserted by the *CryptIC*). Refer to the Analog Devices ADSP-2100 Family User's Manual for more information on 'Bootloading through the IDMA port'.

ACRONYMS/TERMS

25

ACRONYMS

The following table is a list of acronyms used with the description of the co-processor.

Acronym	Meaning
ASIC	<i>Application-Specific Integrated Circuit</i>
CGX	<i>CryptoGraphic eXtensions</i>
CT	<i>Cipher Text</i>
DES	<i>Data Encryption Standard</i>
D-H	<i>Diffie-Hellman</i>
DMA	<i>Direct Memory Access</i>
DSA	<i>Digital Signature Algorithm</i>
IV	<i>Initialization Vector</i>
KCR	<i>Key Cache Register</i>
KEK	<i>Key Encryption Key</i>
LSV	<i>Local Storage Variable</i>
MAC	<i>Message Authentication Code</i>
PCDB	<i>Program Control Data Bits</i>
PIN	<i>Personal Identification Number</i>
PT	<i>Plain Text</i>
SHA	<i>Secure Hash Algorithm</i>
SHS	<i>Secure Hash Standard</i>

TERMS

The following table is a list of terms used with the description of the co-processor.

Term	Meaning
BLACK Key	<i>A secret/private key that is encrypted or covered by a KEK, it can be securely given to another party.</i>
Covered Key	<i>A secret key that has been encrypted, via a KEK, to protect the key from being seen by an undesirable party. Similar to a Black key.</i>
Key Cache Register (KCR)	<i>A working storage area for secret keys, addressable via a register ID between 0 and N.</i>
Key RAM (KRAM)	<i>A volatile public key work area. The public key will be lost during a power-down or reset.</i>
Local Storage Variable (LSV)	<i>A non-volatile Laser-programmed secret key that can be used by the application as its own unique private key. Each CryptIC has a unique LSV programmed into it at the factory.</i>
Program Control Data Bits (PCDB)	<i>Programmable control bits to customize the secure Kernel features (such as allowing RED key exportation/importation, LSV changes, exportable chip, etc.).</i>
RED Key	<i>A secret/private key that is not encrypted or covered by another KEK. It is in its raw unprotected form.</i>

The following is a detailed description of the cryptographic co-processor taken from a CGX interface programmer's guide prepared by the assignee and owner of the invention, Information Resource Engineering, Inc. (IRE). As with the previous description taken from the user's guide, the cryptographic co-processor is often referred to herein by the trademark '*CryptIC*'. The co-processor is also often referred to by part number ADSP 2141. The following includes a detailed description for software developers intending to use the CGX library embedded in the CryptIC. It includes an overview of the CryptIC's CGX software architecture, its key management structure, the command interface, and a detailed description of CGX commands. Incorporated herein by reference is also the Analog Devices *ADSP-2100 family User's Manual* which includes further information on programming the DSP embedded within the *CryptIC*.

GENERAL DESCRIPTION

The CryptIC device is designed to be a highly integrated, general purpose 'security system on-a-chip'. It combines a set of high-performance Hardware blocks to accelerate time consuming cryptographic operations, together with an embedded Digital Signal Processor (DSP) which serves as a crypto system controller. The DSP:

- Implements a 'Security Kernel' which simultaneously enforces certain security policies within the CryptIC and insulates Applications from the details of many complex cryptographic operations.
- Can be used as a security co-processor to manage data movement between a Host system and the CryptIC.
- Can be used to run general-purpose User applications such as a V.34 modem or a Network Interface Card controller.

The Security firmware which is mask-programmed into ROM on the CryptIC is designated as the CGX (CryptoGraphic eXtensions) Kernel. It is a suite of

approximately 40 functions which are available to Applications which require security services.

A protection model is built into the DSP so that security-related and real-time intensive applications can coexist. At any instant in time, the DSP can either be
5 operating in '*Kernel Mode*' which means that one of the CGX commands is in process or it can be in '*User Mode*' which means that a user application is running. In *Kernel Mode*, all of the CryptIC resources are available to the CGX firmware. In *User Mode*, access to Key Storage locations and many of the Security Blocks is restricted in order to enforce proper security policy. A degree of multi-tasking can be achieved between
10 User and Kernel processes, and *Kernel Mode* processing can be interrupted by User code. In addition, a subset of the CGX commands may be preempted by another CGX command.

To simplify Application-level access to crypto functions and allow the coexistence of general-purpose applications running on the DSP, an Application
15 Programming Interface (API) is provided to the CGX Kernel. The CGX Command Interface defines the boundaries between the security functions (which the CGX Kernel implements) and the externally running applications, either on the CryptIC's DSP or on external Hosts communicating via the PCI/PCMCIA bus. A block diagram of the CGX software interface is shown in Figure 11.

CGX SOFTWARE OVERVIEW

One of the primary goals of the *CryptIC* CGX software is to abstract the hardware blocks of the *CryptIC* from the application in a secure and efficient manner. The CGX Kernel has been designed so as to avoid the common problems of linking-in
25 security software with an application or worrying about what resources the application must set aside to accommodate the security software.

The CGX interface is designed so that it can be viewed in one of two ways, depending on the preference of the application programmer. The actual CGX interface

is the same for both views; the difference is in how the application utilizes the CGX interface.

- It can be viewed as a Crypto Library with a C-structure like interface with argument and pointer-passing
- 5 • It can be viewed as a Hardware-accelerated subsystem of the chip, with a register-based interface

In the hardware-oriented view, the CGX interface appears as a 21-word register set (the *command block*) which is mapped into an application-specified memory space in the *CryptIC*. This is similar to the concept of the MMX (Multi-Media eXtension) instructions enhancements to Intel's Pentium chips. The *CryptIC* 10 can similarly be viewed as a single-chip general-purpose processor with added CGX (CryptoGraphic eXtensions) instructions.

To implement the CGX command set, both hardware and software components are integrated on the *CryptIC*. The hardware components are essential in providing 15 the crypto acceleration and for the protection/security of the CGX Kernel and key material from the user application and the external world. As shown in Figure 12, the *CryptIC*'s hardware components are accessed mainly in the Crypto Library, and some are also used in the CGX Command Processor to implement protection features.

The CGX software resides within the dashed line illustrated in Figure 12. The 20 application runs on the DSP and uses the CGX Command Interface as an API to access the CGX command set. To better understand the software architecture of the *CryptIC* security software, a description of each layer is provided in the sub-sections below.

25 Application Layer

The application layer is where the actual application program and data space resides. The application has full-control of the general purpose DSP and its associated

hardware resources. In order to access the cryptographic services, the application must invoke the command interface and pass-in a command code and arguments. The application running in the User space of the DSP can implement anything from a router security co-processor to a V.34 modem data pump.

5 Residing as part of the application layer are the macro functions which IRE provides in its `cgx.h` file. These macros assist the application in preparing the command messages prior to calling the CGX kernel.

CGX Command Interface Layer

10 The CGX command interface layer is an Application Programming Interface (API) which defines the boundaries between the application and the CGX Kernel. The CGX command interface provides the hardware mechanism to enter and exit the CGX Kernel to execute a specific cryptographic command. Part of the command interface function invokes the kernel protection logic to isolate the CGX Kernel and its
15 associated security resources from the external application via the Kernel Protection Logic built into the *CryptIC*. This prevents the possibility of leaking Red key material or of unauthorized external access to the CGX Kernel's program or data space.

 The software interface to the CGX Kernel is via a data structure called the *kernel block*. The *kernel block* is split into two fields: the *command block* and *status block*. The *command block* is used to request a specific cryptographic command and
20 to provide a means to pass-in arguments. The *status block* provides the application with a means to track the status of the CGX Kernel (i.e. is it active or idle) and a means to determine the result status of a requested cryptographic service.

 Therefore, all communications between the application and CGX Kernel is via
25 the command interface and a *Kernel block*. The command interface is discussed in more detail in the section Command Interface.

CGX Command Processor Layer

The CGX Command Processor implements a secure Operating System responsible for processing application requests for various cryptographic services. The CGX Command Processor is invoked by the application via a call to the CGX entry address (0x2000 with PMOVLAY=0x000F). Once the CGX Kernel is active, it can process the requested cryptographic function specified in the *kernel block* defined as part of the CGX command interface layer. The CGX Command Processor is responsible for maintaining the security of the internal cryptographic software, key material, and associated security devices.

Like other operating systems, the CGX Command Processor is responsible for time-sharing the security resources. It does this through DSP context management, preemption management, and system integrity management.

DSP Context Management: Once the CGX Command Processor is activated via the call to address 0x2000, it invokes its DSP context management software. The DSP context management software is responsible for saving the application's current context (i.e. the DSPs registers, frame pointer, and stack pointer). This allows the CGX Kernel to use the full DSP register set. Furthermore, the CGX Command Processor must update the application's *status block* to reflect a 'running' state. Once a cryptographic function is complete, the CGX Command Processor cleans up, modifies the application's *status block* with the result of the cryptographic service, restores the applications DSP context, and returns back to the application.

Preemption Management: For certain CGX commands, the Command Processor can allow a new command to preempt a running one. This includes when it is actively processing one of the public key or digital signature commands and the preempting command is a hash or symmetric key command. This feature is provided because some of the public key commands can take hundreds of milliseconds to complete. However, if a preemptive request comes in and the CGX Kernel is not

executing one of the preemptable commands, the CGX Command Processor will return a CGX_BUSY_S status code and will not execute the command.

System Integrity Management: Lastly, the CGX Command Processor is responsible for monitoring the security integrity of the *CryptIC* software and hardware resources. As part of initialization processing, the CGX Command Processor runs a suite of self-tests to verify the health of the security components. The CGX Command Processor will not give control of the DSP to the application until the self-test suite completes successfully. Furthermore, there are several protection mechanisms to prevent key material leakage which require the CGX Command Processor to explicitly set hardware register bits enabling specific cryptographic service. This level of control is only permitted for the CGX Command Processor; thus preventing accidental or intentional access to areas of the security blocks not allowed.

CGX Overlay Layer

The CGX overlay layer is provided as the interface into IRE's CryptoLIB software. The CryptoLIB software is a library that is designed for multiple platforms, ranging from the PC to embedded systems. The CGX overlay acts as the 'wrap code' to enable the library to execute on the *CryptIC* platform unmodified. Figure 13 illustrates the data flow through the CGX overlay layer.

When a cryptographic request is received, the CGX Command Processor parses the *kernel block* to determine the cryptographic command to execute. The CGX Command Processor executes a CGX overlay operation from a table, based on the command value embedded in the *command block* portion of the *kernel block*. The CGX overlay operation is responsible for extracting the arguments from the *kernel block* and invoking the proper CryptoLIB operations. In some cases, the CGX overlay operation may invoke several CryptoLIB operations. In effect, this is an object-oriented approach where the CGX overlay class is the parent class to the CryptoLIB classes.

CryptoLIB Layer

5 The CryptoLIB layer contains IRE's Crypto Library software and it also contains drivers for the *CryptIC* hardware blocks as well as 'soft' versions of the hardware algorithms (the soft versions are ifdef'ed out at compile time for the *CryptIC*). The CryptoLIB software is a library of many cryptographic classes implementing various cryptographic algorithms from symmetrical encryption algorithms to one-way Hash functions, to public key operations.

10 The CryptoLIB API is transparent to whether it is running with hardware acceleration or utilizing the library's software crypto functions. This hides the implementation of the *CryptIC* platform and allows full reuse of the general CryptoLIB software. This provides several advantages including:

- The CryptoLIB software can be independently used in the PC environment
- The software only 'C' version of CryptoLIB can be used during development of products which will use the *CryptIC*.
- 15 • Enhancements and modifications to the CryptoLIB can be made and tested in the PC environment before migrating them into the hardware of the *CryptIC*.

CGX SOFTWARE COMPONENTS

20 This section describes the various software components that make up the *CryptIC* in more detail than the previous section. From the CGX programmer's perspective, it is primarily an architectural reference, since most of this functionality is hidden inside the CGX kernel.

CGX Command Processor

25 The CGX Command Processor is made up of many software components:

- Reset processing

- Initialization processing
- Contention/preemption/reentrance processing
- Configuration storage
- Key storage
- 5 • Command interface
- Context management
- Kernel operations

The following sub-sections describe the components that make up the CGX Command Processor.

10 **Reset Processing**

When the *CryptIC* is first powered-up, or after any Reset event, control is given to the CGX Kernel prior to allowing any application code to execute. The CGX Kernel performs a very quick 'reset-initialization': It first checks a 32-bit INIT flag location in KRAM to determine if the Kernel has previously been initialized. If it has, then the DSP's Computational registers and DAG registers are cleared. If the Kernel has not been initialized, then the DM_Reserve and PM_Reserve registers are cleared in order to free the DSP's internal PM and DM for application use. Finally, the CGX Kernel jumps to address 0x0000 in program memory (PMOVLAY=0) to resume the normal reset sequence.

20 **Initialization Processing**

There are two modes of initialization processing in the CGX kernel: *Full-Init* and *Basic-Init*. Full-Init occurs when the application issues the CGX_INIT command to the CGX kernel. Basic-Init is a safety net which will be automatically run if a CGX command is issued prior to the Full-Init being run.

25 The Full-Init actually executes the Basic-Init processing, but in addition provides some Kernel and Hardware configuration capability. When CGX_INIT is

called, the application may supply a Kernel Configuration String (KCS) and/or a Program Control Data Bits (PCDB) configuration string in order to modify the behavior of the CGX kernel and the hardware functions of the *CryptIC*.

5 At some point after a reset, and the application code has control, the first CGX command is executed, passing control to the CGX Kernel. When the CGX Kernel is entered, it must first determine what type of operation to perform: Basic Initialization or CGX command processing. To determine what to do, the CGX Kernel reads the INIT flag, which is a 32-bit location in Kernel RAM. This flag represent the initialization state of the CGX kernel: If the flag is set to a pre-determined value, then
10 it is an indication that the kernel mode has been previously entered and initialization has occurred, thus the CGX Command Processor proceeds to service the *kernel block* and implement the requested CGX command.

If the flag is not set to the proper value, then the kernel will assume that it first has to perform Basic-Init processing.

15 Note that it is normally expected that the application will first execute the CGX_INIT command prior to requesting any other CGX commands. However, since this cannot be guaranteed, the following processing is included in the CGX entry processing.

Basic-Init Processing

20 If the CGX Kernel sees that the INIT flag has not been set when a CGX command is issued, it first enters the 'Basic-Init' mode to perform self-tests and other initialization steps. Upon returning from Basic-Init, the CGX Kernel sets the INIT flag, assuming the self-tests pass. From this point on, until the next power-cycle, the INIT flag will indicate that the kernel has been initialized and the CGX Kernel will not
25 re-execute the startup self-tests.

As part of the Basic-Init processing, the CGX Kernel executes a set of self-tests to check the integrity of its ROM (using a pre-calculated MAC) Kernel RAM,

and associated security devices (i.e. RNG). If any of these self-tests fail, the CGX Kernel returns a failure code (ie. CGX_RAM_FAIL_S) to the application via the status field of the *kernel block* and does not set the INIT flag. Furthermore, it will not execute any command until the self-tests pass and the INIT flag is written with a valid value.

Once the self-tests complete successfully, the CGX Kernel can proceed with the remainder of its initialization. There are several areas to initialize, the extended RAM/ROM Reserve bits, the PCDBs, and the KCS. Once the Basic-Init is complete, the CGX Kernel will proceed to process the original requested CGX command.

Full-Init Processing

At any time after reset, the initialization mode can be reentered by the application by using the CGX_INIT command. As part of the initialization command, the application must pass in two initialization strings, the PCDB string and the Kernel Configuration String (KCS). The initialization strings are an array of defined bits to allow the application to set, disable, and enable various features of the CGX Kernel (KCS) and of the *CryptIC* hardware blocks (PCDBs). Once the full initialization is complete, the CGX Kernel enters an idle mode and returns to the caller, waiting to service application requested CGX commands received across the command interface.

The following sub-sections define the initialization strings and the areas that are initialized as part of the Basic-Init or upon the application's request for the CGX_INIT command.

PM/DM Reserve Bits

If the initialization mode is entered for the first time, the INIT flag will not match its predefined value. In this case, the initialization routine must clear the PM_Reserve and DM_Reserve registers. These are two 16-bit registers for which each bit represents a 1K memory block of internal PM or DM memory. A bit set to '1'

indicates that the corresponding 1k memory block has been locked into use by the CGX Kernel and cannot be accessed in user mode.

5 These bits are preserved through resets. Therefore, if any of the PM/DM_Reserve bits are set (i.e. a 1) the CGX Kernel must erase the RAM blocks that are locked, since there may be sensitive information in this memory. Then after each locked block is erased, the CGX kernel can release them by clearing the PM & DM Reserve bits to a 0.

Programmable Control Data Bits (PCDB) Initialization

10 The PCDBs are laser-programmed at the factory with a set of defaults and may later be changed by the application via a special unlock-data token that IRE or a designated OEM can provide. The end-user can use an unlock-data token provided by IRE to later enable/disable various features via the initialize command (i.e. CGX_INIT). Managing the access to the PCDBs with an unlock data message allows IRE or other OEMs to control what type of services customers have access to. The PCDB bits must be initialized if not already done. Initially, the laser-burned copy of the PCDBs are moved to the working RAM copy. However, by invoking the CGX_INIT command, the application can re-program some of the RAM shadowed PCDB bits, providing a digitally-signed token is presented along with the request.

20 If the application desires to later change the PCDBs back to the factory laser default, the application can use the restore-default command, CGX_DEFAULT.

25 The PCDBs allow the application to customize the CGX Kernel's cryptographic operations and control. The PCDBs contain such items as: a bit to allow importing Red KEKs, enabling/disabling crypto-algorithms to make a device exportable, configuring symmetric and public key lengths, etc. Refer to the section Kernel Configuration String (KCS) for a detailed description of the PCDB bits and their meanings.

Contention, Preemption, and Reentrance Processing

The CGX Kernel is preemptive, and for certain commands it is also reentrant. Being preemptive means that if an interrupt occurs while the CGX Kernel is active, it allows the interrupt to vector to the application's interrupt service routine (ISR) in 'User memory'. Then after the interrupt has been completely processed, the CGX Kernel is resumed via the application's Return From Interrupt (RTI) instruction; thus the preempted CGX operation is continued.

For certain CGX commands (ie. public key and digital signature), it is possible to interrupt the command while it is running and then, within the interrupt service routine, issue another CGX command. The CGX Kernel firmware maintains two stacks and workspaces in order to allow 1 level of preemption/reentrance. The CGX commands are shown in the table below.

CGX Command	Can Preempt a CGX Command	Can be Preempted by Another CGX Command
CGX_INIT		
CGX_DEFAULT		
CGX_RANDOM	<input type="checkbox"/>	
CGX_GET_CHIPINFO	<input type="checkbox"/>	<input type="checkbox"/>
<i>Symmetric Key Commands</i>		
CGX_UNCOVER_KEY	<input type="checkbox"/>	
CGX_GEN_KEK	<input type="checkbox"/>	
CGX_GEN_KEY	<input type="checkbox"/>	
CGX_LOAD_KEY	<input type="checkbox"/>	
CGX_DERIVE_KEY	<input type="checkbox"/>	
CGX_TRANSFORM_KEY	<input type="checkbox"/>	
CGX_EXPORT_KEY	<input type="checkbox"/>	
CGX_IMPORT_KEY	<input type="checkbox"/>	
CGX_DESTROY_KEY	<input type="checkbox"/>	
CGX_LOAD_KG	<input type="checkbox"/>	

CGX ENCRYPT	<input type="checkbox"/>	
CGX DECRYPT	<input type="checkbox"/>	
<i>Asymmetric Key Commands</i>		
CGX GEN PUBKEY		<input type="checkbox"/>
CGX GEN NEWPUBKEY		<input type="checkbox"/>
CGX GEN NEGKEY		<input type="checkbox"/>
CGX PUBKEY ENCRYPT		<input type="checkbox"/>
CGX PUBKEY DECRYPT		<input type="checkbox"/>
CGX EXPORT PUBKEY		<input type="checkbox"/>
CGX EXPORT PUBKEY		<input type="checkbox"/>
<i>Digital Signature Commands</i>		
CGX SIGN		<input type="checkbox"/>
CGX VERIFY		<input type="checkbox"/>
<i>Extended Algorithm Commands</i>		
CGX LOAD EXTENDED		
CGX_EXECUTE_EXTENDE D	<input type="checkbox"/>	<input type="checkbox"/>
<i>Hash Commands</i>		
CGX HASH INIT	<input type="checkbox"/>	
CGX HASH DATA	<input type="checkbox"/>	
CGX HASH ENCRYPT	<input type="checkbox"/>	
CGX HASH DECRYPT	<input type="checkbox"/>	

<i>Prf Commands</i>		
CGX PRF DATA	<input type="checkbox"/>	
CGX PRF KEY	<input type="checkbox"/>	
CGX MERGE KEY	<input type="checkbox"/>	
CGX MERGE LONG KEY	<input type="checkbox"/>	
CGX LONG2BLACK	<input type="checkbox"/>	
<i>Key Recovery Commands</i>		
CGX GEN RKEK		
CGX SAVE KEY		
<i>Math Commands</i>		
CGX MATH		<input type="checkbox"/>

Table 10 *CGX Command Preemption*

If an application attempts to perform a CGX preemption (ie. 2nd call into CGX) which violates the above policy, or if it attempts to preempt a CGX command with another one while the Kernel is already running a 2nd command, then the CGX call will return quickly with a CGX_BUSY_S error code in the *Status block*. In this case, the application will have to return from interrupt in order to let the currently running CGX command complete before attempting another CGX call.

To aid the application in determining the state of the CGX Kernel, a simple contention management scheme is available to the application. Contention management is necessary in order to prevent the application from attempting to re-enter the CGX Kernel while it is in a preempted state. This is a simple problem from the stand-point of the CGX Kernel because internally it can maintain semaphores to determine if preemption of an active command is allowed or whether it is already one level deep, executing a 2nd CGX command. To easily allow the application to determine the CGX Kernel's state, the status of the CGX Kernel (i.e. active, complete, etc.) can be queried by the application at any time. The state can be obtained by looking at the current *status block* in DSP internal or external memory (the *status block* is part of the *kernel block*, as explained later).

This scheme has three advantages: First, the application does not have to invoke a command to query the state of the CGX Kernel, it can be read directly from the *status block* in memory. Secondly, it removes the complexity and burden of the application keeping track of the CGX Kernel's state on its own. Lastly, the *status block* provides the means for the CGX Kernel to retrieve the status result of the CGX operation (success or error code) which can be used by the application to debug/test the cryptographic operations.

Saving DSP Context at Interrupt

When a CGX operation is interrupted by the application, certain elements of the DSP processor's context must be saved and then restored upon return-from-interrupt. (This is of course necessary with any interrupt service routine.) The DSP hardware automatically saves the Program Counter on the hardware PC Stack and the Status Registers on the Status Stack. It is up to the application to determine if it should pull these values off of the hardware stack and move them to an application software stack.

In addition, the interrupting code must save the DAG registers, the PX register (lower 8-bits of Program Memory reads) and PMOVLAY, DMOVLAY registers as well as the computational registers (2 banks). In order to optimize interrupt processing, the application code can be designed to run in the Primary register set, while the CGX code can execute in the Secondary set. In this case, the application code will have to insure that the register set selection (MSTAT register) is properly handled upon interrupt and return-from-interrupt.

Once the external interrupt service completes, it restores the CGX Kernel context and invokes the RTI operation. The Status registers and Program Counter are automatically popped off the hardware stack at the RTI instruction. The return-from-interrupt operation vectors back into the CGX Kernel mode, at which point the Kernel Protection logic will resume and the interrupted CGX command processing will continue.

Storage Management

The CGX Command Processor is responsible for the storage devices associated with the security block of the *CryptIC*. The storage areas that concern the CGX Command Processor are the CGX Kernel's KRAM, the volatile Key Cache Registers, and the PM/DM_Reserve RAM blocks.

Furthermore, for the CGX Command Processor to fully protect key material and algorithms, it provides a protection of the DSP's registers as well as its RAM space. Although the RAM space is protected via specialized hardware, the CGX Command Processor uses critical regions (interrupts disabled) to protect registers. For example, when key material is moved in or out of the Kernel RAM, the DSP's registers are used. Therefore, a potential to read the key material exists should an intruder try to single-step the CGX Kernel via interrupts. Although the application can not read the protected KRAM and kernel ROM, it could read the DSP's registers. Therefore, all key material movement is done via the mem_cpy operation. This operation copies data from one place to another with interrupts disabled, and at the end it sets the registers it used to 0. This prevents applications from single stepping the Kernel. The same is done to the pm_cpy operation which is used to read program memory.

As mentioned earlier, the hardware provides key aspects to protecting various memory stores. In particular, the hardware provides several bus transceivers to only allow one memory to be accessed at a time. So that when the external RAM/ROM is accessed the external application can not read any of the CGX Kernel's internal memory (KRAM, kernel ROM, and extended RAM/ROM). Furthermore, the bus transceivers are placed on some of the security block memory devices in order to prevent the CGX Kernel from accidentally mixing Red key material with other memory devices used by the CGX Kernel. The following sub-sections outline the areas where the CGX Kernel enforces the security model.

Volatile Key Storage

The volatile key storage houses the private portion of a public key in the public key area and a fixed number of secret keys in the Key Cache Register (KCR) area. The volatile key area is also referred to as the actively working keys. All cryptographic commands operate only on the active volatile working keys. The application is responsible for moving keys between the outside world and the KCRs.

The CGX Command Processor only allocates enough space in the volatile KRAM for 15 symmetric Key Cache Registers. The CGX Command Processor provides ease of access to the secret keys by allowing them to be referenced by a register ID with numbers from 0 through 14.

Extended (Reserved) RAM Blocks

Under various circumstances, one or more 1kword segments of PM and/or DM memory can be locked-into the kernel space. For example, as part of the CGX_INIT processing, the application can request to allocate some of the DM towards extending the number of Key Cache Registers.

The CGX Command Processor takes care of setting and resetting the PM_Reserve and DM_Reserve bits to lock and unlock blocks of memory. It also manages the secure information which is stored in these blocks and insures that if a block is unlocked, that its data is first erased.

Command Interface Processing

The CGX Command Processor is responsible for managing the command interface between the application and itself or the cryptographic services. The command interface to the CGX Kernel is accomplished using a shared memory block and a special transfer address mechanism. The shared memory block (*kernel block*) provides the means for the application to communicate with the CGX Kernel, allowing the application the ability to query the CGX Kernel's status and request cryptographic

services. The special transfer mechanism (CGX Kernel transfer routine) allows the application to invoke the CGX Kernel. The command interface is described in detail in the section Command Interface.

Command Processor Micro-Operations

5 The CGX Command Processor is similar to a command interpreter; it consumes the command and its arguments, invokes the proper service to execute the command, and returns the result. However, the CGX Command Processor is a little more complicated than that since it must perform several more micro-tasks similar to an OS. It must perform context switching, I/O services, and monitor the security,
10 integrity, and health of the internal hardware components.

The CGX Command Processor is comprised of three modules of software: the entry code, the cryptographic service executor, and the exit code.

Entry

15 When the transfer vector is initiated by the application, the CGX Kernel is entered by setting the PMOVLAY register to 0x000F and then executing a CALL 0x2000 instruction. From the call instruction and some associated hardware logic and signals, the internally ROM encoded CGX Kernel program block is overlaid in the upper portion of the DSP's program space. This allows the CGX Kernel code to become active and transparent to the application. When the ROM-based CGX Kernel
20 program block is overlaid, the processor immediately branches to the CGX Command Processor entry handler. The purpose of the CGX Command Processor entry handler is to determine the type of service it must perform. The entry code is all written in assembler due to real-time constraints of context saving and service restoration in the case of a command preemption.

25 The entry code runs with all interrupts disabled upon entry. The interrupts remain disabled for just for the entry code, until a point at which the CGX kernel has verified that it can perform the operation and has saved the application's context.

Once the entry code is operational, it sets up a local stack to execute from, and the CGX Command Processor saves the context of the externally running application (i.e. saves registers, the PC return address, the stack pointer, frame pointer, etc.), assigns a global pointer to the *command block*, writes a CGX_RUNNING_S code to the external *status block* (i.e. the CGX Command Processor is active),
5 reenables interrupts, and calls the cryptographic command interpreter (described in the next sub-section).

However, if the CGX Command Processor is already active, it sets up a second local stack to execute from, (i.e. it is now being preempted) it saves the context of the
10 externally running application (i.e. saves registers, the PC return address, the stack pointer, frame pointer, etc.), it saves a context of the preempted cryptographic command (i.e. save the global pointers to the *command/status blocks*, etc.), assigns a global pointer to the *command block*, writes a CGX_RUNNING_S code to the external *status block* (i.e. the CGX Command Processor is active), reenables
15 interrupts, and calls the cryptographic command interpreter (described in the next sub-section).

Cryptographic Service Execution

The cryptographic command interpreter is fairly simple. The code accesses the *command block* pointer established in the entry code and based on the command, it
20 invokes a cryptographic operation from a table of CGX overlay operations. No arguments are passed into the overlay operations. The overlay operations use the *command block* pointer to get access to the argument list. Figure 14 illustrates this hierarchical approach to accessing the cryptographic operations.

The purpose of the layered hierarchical approach is simply to isolate the
25 knowledge each layer needs to operate within the CGX Kernel. This allows the lowest level, the CryptoLIB to only have to implement cryptographic operations. To make this work, a middle layer called the CGX overlay is provided. The middle layer

actually implements the wrap code around the cryptographic library, enabling memory, accessing the key RAM, etc..

As Figure 14 shows, the overlay operation is responsible for invoking the CryptoLIB operations in order to carry out the requested cryptographic service. The overlay wrap code extracts the inputs and outputs from the *command block* and passes them as arguments to the CryptoLIB operations. Furthermore, the overlay operations, which are actually CGX Kernel operations, are responsible for enabling write access to external RAM and for updating the status field of the *status block*. This isolates the control of the sensitive hardware access features to the CGX Kernel only. Once the overlay operation is complete, the exit code is invoked.

Exit

The exit handler of the CGX Command Processor performs the following:

- disable interrupts to establish a critical region,
- restores the external application's DSP context,
- writes the result code to the *status block*,
- re-enables interrupts, and
- cause the transfer back into the application and user memory space.

Also, as a security precaution, the write access enable to external RAM is disabled before returning to the application.

CGX Overlay

The purpose of the overlay operations is two fold: first, it provides a standardized interface to IRE's CryptoLIB. The overlay operations contain some wrap code and calls to CryptoLIB operations. The second purpose is to allow the reuse of IRE's CryptoLIB intact. Without the overlay operations, the CryptoLIB code would have to understand the *kernel block* interface; thus creating a non-standard package. Instead, a clean implementation of the CryptoLIB, using a C interface, is possible.

There is a CGX overlay operation for every cryptographic command provided by the command interface. The CGX operations reside in a single table that is accessed by the CGX Command Processor software by the command value embedded in the *command block* of the *kernel block*. Part of this table are bits that represent the
5 preemption requirements and the privilege access bits.

CryptoLIB

The CryptoLIB software is a library of software functions that implement a full suite of cryptographic services. It is designed to be portable and flexible and the
10 interface allows for certain functions to be customized by the application.

The interface to the CryptoLIB follows the ANSI C standards, and the CryptoLIB is all written in ANSI C format. The reason behind this is so that the same library base can be used in desktop environments as well as embedded applications like the *CryptIC*.

15 Therefore, since no special modifications to the CryptoLIB have been made, the CGX Kernel is recognized as another application to the CryptoLIB software. The CGX Kernel uses the library calls as would any other application in a PC environment or another embedded environment would. This has been accomplished because of the CryptoLIB's ability to allow it to be customized by the application.

Hardware / Platform Specific Drivers

A number of *CryptIC* specific drivers are provided. The drivers provided are:

- a BIGNUM hardware assist driver,
- a random number generator driver, and
- 25 • crypto-block operations (i.e. DES engine).

SELF-TESTS

The self-test routines provide an integral part of determining the integrity and health of the security blocks. The self-test operations are used as part of the CGX Kernel reset processing to verify the health of the security blocks. The following is a list of self-tests that are supported:

- CGX Kernel code integrity check,
 - This self-test checks the integrity of the CGX Kernel's program by running a MAC over its entire 32K program space and comparing the MAC to the ROM encoded MAC.
- randomizer check,
 - This self-test checks the operation of the hardware randomizer. The method for this test is described in the FIPS 140-1 document for 'Continuous Random Number' test.
- volatile key storage checks
 - This self-test checks the volatile key storage (i.e. internal KRAM). The purpose is to verify the key storage integrity. To do this read, write, and stuck-at checks are performed.

SOFTWARE STATES

The CGX Kernel is driven by a state machine as shown in Figure 15. As can be seen in the figure, the state machine is fairly simple, comprising only four states.

RESET State

The RESET state can be transitioned to from any one of the four states, including the RESET state. The RESET state is entered upon powering up of the *CryptIC*, through the command of the application, or a fatal error detected by the CGX Kernel.

As discussed in the section Reset Processing, the RESET state performs the Basic-Init functions which include testing the sanity and integrity of its services, resources, and program. If any of these items fail, the CGX Kernel will reset the processor and the RESET state will be re-entered.

5 At the successful completion of self-tests the CGX Kernel will transition to the IDLE state. The only other state the CGX Kernel can transition to is the RESET state.

IDLE State

10 The IDLE state can be transitioned to from either the COMMAND state or COMMAND_BLOCK state. The IDLE state is entered upon the successful completion of the CGX Kernel's reset self-test suite or after the completion of a cryptographic command requested by the application.

15 In the IDLE state the CGX Kernel is inactive, it is idly waiting for the application's request for the next cryptographic command. Upon the request of a cryptographic command the CGX Kernel will transition to the RESET, COMMAND, or COMMAND_BLOCK state. The CGX Kernel will transition to the RESET state if an invalid *kernel block* is received. The CGX Kernel will transition to the COMMAND state if the preemption bit in the control field in the CGX overlay tuple data structure (*cgx_overlay_tuple*) is set to a 0; otherwise if the bit is set to a 1, it
20 transitions to the COMMAND_BLOCK state. The preemption bit is described in the section CGX Overlay Interface.

COMMAND State

25 The COMMAND state can be transitioned to from either the COMMAND_BLOCK state or IDLE state. The COMMAND state is entered upon a request from the application of a cryptographic command; either back from preemption or an initial application request. When in the COMMAND state, the CGX Kernel

allows the current cryptographic command to be preempted. However, only certain commands can be preempted and only certain commands can cause the preemption.

5 All of the public key and digital signature operations have the preemption bit set to a 0; thus allowing them to be preempted and causing the CGX Kernel to enter the COMMAND state from the IDLE state. A number of other CGX commands are allowed to preempt the public key and digital signature algorithms (see section 0).

Furthermore, the CGX Kernel only allows one level deep of preemption. Therefore, upon preemption the CGX Kernel transitions into the COMMAND_BLOCK state to prevent further preemptions. Thus, the cryptographic
10 operation that caused the preemption must run to completion before the original one can complete or another one can preempt.

At the completion of the requested cryptographic command the CGX Kernel transitions back to the IDLE state. However, if some sort of severe error or hardware failure occurred the CGX Kernel will reset the processor; thus transition to the RESET
15 state.

COMMAND BLOCK STATE

The COMMAND_BLOCK state can be transitioned to from either the COMMAND state or IDLE state. The COMMAND_BLOCK state is entered upon
20 request by the application of a cryptographic command; either through preemption out of the COMMAND state, or via a single application request of a non-preemptable cryptographic command. When in the COMMAND_BLOCK state, the CGX Kernel will not allow the current cryptographic command to be preempted.

At the completion of the requested cryptographic command, the CGX Kernel
25 transitions back to the IDLE state if it was not previously entered due to preemption. If the CGX Kernel entered the COMMAND_BLOCK state because the application requested a CGX command that caused an active CGX command to be preempted, the CGX Kernel will transition back into the COMMAND state to complete the original

CGX command. However, if some sort of severe error or hardware failure occurred, the CGX Kernel will reset the processor; thus transitioning to the RESET state.

KEY MANAGEMENT

INTRODUCTION

It is well known that a carefully designed and implemented Key Management infrastructure is a critical component of a good security system. In keeping with the philosophy of a 'security system on-a-chip', the *CryptIC* incorporates a powerful and secure key management system into both its hardware and CGX firmware.

There are generally 5 elements of key management to consider:

- 1) Key Generation
- 2) Key Distribution
- 3) Key Storage
- 4) Key Selection (and use)
- 5) Key Destruction

The CryptIC's features support a consistent flow of Key Material through all of the five phases. In particular:

- The facilities are provided by the CGX library (and the integrated hardware blocks) to generate and safely store key material.
- Key 'covering' with a Key Encryption Key (KEK) is a CGX service which allows secure key distribution and secure storage.
- Key selection is consistently handled through CGX library arguments and safeguards are implemented against key hijacking, spoofing, alteration, etc..
- Key destruction may easily be achieved either by powering-off the *CryptIC*, or by executing a Destroy_Key CGX command.

The *CryptIC* allows a wide range of key management implementations, since it is only concerned with supplying the primitive key management utilities. This allows the application using the *CryptIC* to create either a simple 'flat' key management structure, or a highly layered and complex 'military-grade' key management system.

The following sections describe the types of keys used on the *CryptIC*, the key handling and usage conventions, and finally a discussion of key storage requirements and definitions.

KEYS

The *CryptIC* supports the following algorithms and key sizes for Symmetric (secret key) and Asymmetric (public key) cryptographic operations, shown in the table below:

SYMMETRIC ENCRYPTION:	KEY SIZES	In Steps Of
DES	40 – 56 bits	8 bits
Triple-DES (two key)	64 – 112 bits	8 bits
Triple-DES (three key)	120 – 168 bits	8 bits
PUBLIC KEY ENCRYPTION:		
Diffie-Hellman	512 – 2048 bits	64 bits
RSA *	512 – 2048 bits	64 bits
DIGITAL SIGNATURES:		
DSA	512 – 2048 bits	64 bits
RSA *	512 – 2048 bits	64 bits

* requires RSA licensed version of CryptIC

Table 11 Algorithms and Key Sizes

Since the *CryptIC* implements multiple cryptographic algorithms, there is a need for several types of key. The following descriptions of the keys are grouped into **Symmetric** keys (also known as secret keys) and **Asymmetric** keys (or public key pairs).

SYMMETRIC KEYS

Symmetric keys are used in order to support the symmetric cryptographic algorithms (i.e. DES, Triple DES). The *CryptIC* supports several symmetric key types

for data encryption and key encryption (covering/covering). In addition to describing the three types of symmetric keys, there are several other issues discussed in this section. These include: key length, key generation, key access, and key protection.

Symmetric Key Types

5 There are three types of symmetric keys used on the *CryptIC*:

- Data Encryption Keys (DEKs)
- Key Encryption Keys (KEKs)
- Message Authentication Code keys (HMAC).

10 Each of the keys are described along with their respective properties in the following subsections.

Data Encryption Keys (DEKs)

Data encryption keys (otherwise known as session keys or traffic keys) allow a message, a communications channel, a data file, etc. to be encrypted using one of the symmetric algorithms (i.e. DES, triple-DES, etc.) supported by the *CryptIC*.

15 All DEKs are stored in and retrieved from volatile Key Cache Registers (KCRs) or written to the encrypt hardware engine's context_0 or context_1 data key register. By default, a DEK is not allowed to be exported (i.e. read by the application) unless it is covered by a key encryption key. However, the DEK must be uncovered in the *CryptIC* device to be used as a traffic key.

20 A DEK can be generated via all the means described in the section Symmetric Key Generation, or can be imported as a Red key that was generated by an external application (assuming the applicable Program Control Data Bit allows Red key load).

25 Typically, DEKs are short lived (ephemeral); they are around long enough to encrypt the active secure channel. Once the secure channel is no longer required, the DEK is destroyed. The CGX Kernel that controls the cryptographic operations on the *CryptIC* is not responsible for destroying DEKs since it doesn't know when a secure

channel has terminated. This is left to the application via the CGX_DESTROY_KEY command. However, the CGX Kernel will destroy all volatile key cache locations upon a reset or protection violation.

Key Encryption Keys (KEKs)

5 A key encryption key (otherwise known as a covering key) allows keys to be off-loaded from the CryptIC for storage (ie. power-down recovery, key escrow, etc.), to allow key distribution, or for the secure exchange of traffic keys. The *CryptIC* supports five types of KEKs:

- Local Storage Variable (LSV)
- 10 • internal Generator Key Encrypting Key (GKEK)
- one or more application Key Encrypting Keys (KEKs)
- one or more Hash/Encrypt Data Key protection KEKs (DKEKs)
- optionally a Recovery KEK (RKEK)

15 Only the LSV is preserved across resets since it is laser-stored within the CryptIC. Therefore, the storage requirements for other KEKs must be implemented by the application. This would be achieved by off-loading the KEK covered by another KEK. (The LSV is used to cover the RKEK and GKEKs. GKEKs or other KEKs are used to cover application KEKs, DKEKs and DEKs.) The KEK heirarchy is shown in Figure 16.

20

Local Storage Variable (LSV)

25 The LSV is a non-volatile 112-bit KEK which is laser-burned into each *CryptIC* device at fabrication. It is unique to each *CryptIC* produced and can be considered that chip's master 'root' key. Only GKEKs and the RKEK may be covered by the LSV.

Internal Generator KEK (GKEK)

The GKEK allows the application to safely build its own symmetric key hierarchy trees securely by not allowing any of the user key's Red material to be exposed. The GKEKs allow for the off-loading of internally generated user keys to be stored in the application's non-volatile store. The GKEKs effectively insulate the LSV from any direct attack, since GKEKs are always trusted and cannot be imported or exported.

Application-created KEKs

Applications can generate multiple KEKs by calling upon the CGX library to generate a KEK and then request the *CryptIC* to use the generated key to cover/uncover other keys. The application must off-load or export the KEK and store it for later use. This means the application must first cover the newly generated KEK with either the GKEK or another KEK so that it can leave the *CryptIC*. This approach offers a layered model of KEKs.

Hash/Encrypt Data key protection KEKs (DKEKs)

DKEKs are used exclusively to cover and uncover DEKs which will be used in the Hash/Encrypt hardware engine. This KEK will typically be written into the hardware context 0 or context 1 DKEK register so that it can automatically decrypt 'Black' DEKs which may be stored in a crypto context database off-chip. DKEKs cannot encrypt other KEKs.

Recovery KEK (RKEK)

A single RKEK is supported in the *CryptIC*. The RKEK is always a Diffie-Hellman derived key and requires an IRE-certified request to generate it. It may be off-loaded from the *CryptIC* covered by the LSV. The RKEK is used for covering DEKs, KEKs, or DKEKs for escrowing off-chip or for Key Recovery implementations.

HMAC Keys

An HMAC key is a special type of key which does not impose formatting constraints. An HMAC key type may be used to store the secret output of a hash result – perhaps as part of a key derivation process, or it may be used to store a secret value to be used as the input to an HMAC function. HMAC keys may be from 40 to 160-bits long in 8-bit steps.

Since it is un-formatted, key weakening is never applied to an HMAC key.

Symmetric Key Lengths

The *CryptIC* supports several key lengths depending on the symmetric block algorithm and the type of device (i.e. domestic version versus exportable one). The key length can be adjusted between the range of 40 bits and 168 bits, depending on the PCDB programming within the *CryptIC* (i.e. configured for domestic or export).

For a domestic *CryptIC* all DES and Triple DES keys can have a 40 bit to 168 bit key length, programmable in 8-bit increments by the application. This allows for variable key lengths other than the typical 40, 56, 112, and 168-bit key lengths. The *CryptIC* defines triple DES keys as three 56 bit DES keys (i.e. 168 bits in total), however a triple DES key will actually occupy 192 bits (i.e. 24 bytes) since in DES, one bit of each key byte is considered a parity bit. To allow for non-standard size keys, a symmetric key weakening algorithm is used and described below in this section.

Any key which is specified to be 40, 48 or 56-bits will automatically be considered a single-DES key. Any key which has an effective length of 64 to 112 bits will become a two-key, triple-DES key. That is, the significant bits of the key will be distributed between the first and second 56-bit sub-keys, and then the third sub-key will be a copy of the first. Any key with an effective length of 120 to 168 bits will become a three-key, triple-DES key.

This means that any key longer than 56 bits is represented in a triple DES form – ie. three 56-bit DES keys. The keys are always expanded to the full 168-bits in a manner to reflect the number of bits chosen by the application.

Symmetric Key Generation

- 5 Symmetric keys can be generated six ways on the *CryptIC* as described in the table below:

KEY GEN Technique	CGX Call	Description
Generate DEK from RNG	CGX_GEN_KEY	Samples the output of the Random Number Generator to assemble the desired length DEK
Generate KEK from RNG	CGX_GEN_KEK	Samples the output of the Random Number Generator to assemble the desired length GKEK
Negotiate Diffie-Hellman	CGX_GEN_NEGKEY	Perform the Diffie-Hellman G^x exponentiation in order to arrive at a shared secret value. Based on ANSI X9.42.
Hash from Password/Passphrase	CGX_DERIVE_KEY	Derive a symmetric secret key by hashing an application-supplied Password or Passphrase.
Transform key (i.e. IPsec)	CGX_TRANSFORM_KEY	Transform a key using a combination of Hashing, Mixing with fixed data and re-Hashing, XORing, etc.
Import a Red key from the application	CGX_LOAD_KEY	Import a Red key provided by the application.

Table 12 Key Generation Techniques

Symmetric Key Representation

- 10 Symmetric keys are represented in three ways, an **Internal form**, an **IRE External form**, and a **Inter-operable External form**. The first two representations are used in order to enforce the security policy of the *CryptIC*, and the last is used in order to allow the *CryptIC* to share key material with other vendor's implementations.

Symmetric Key IRE Internal Representation

- 15 Symmetric keys are stored within the *CryptIC* in Key Cache Registers (KCRs). Note that all of the data stored is in a plaintext (Red) form so that the CGX software may access it without the overhead of decrypting anything. Untrusted software running in the 'User Mode' of the DSP does not have access to the KCRs.

The format of the storage is as follows:

[type][keylength][key-bits][weakened_keybits][keycount][attributes]

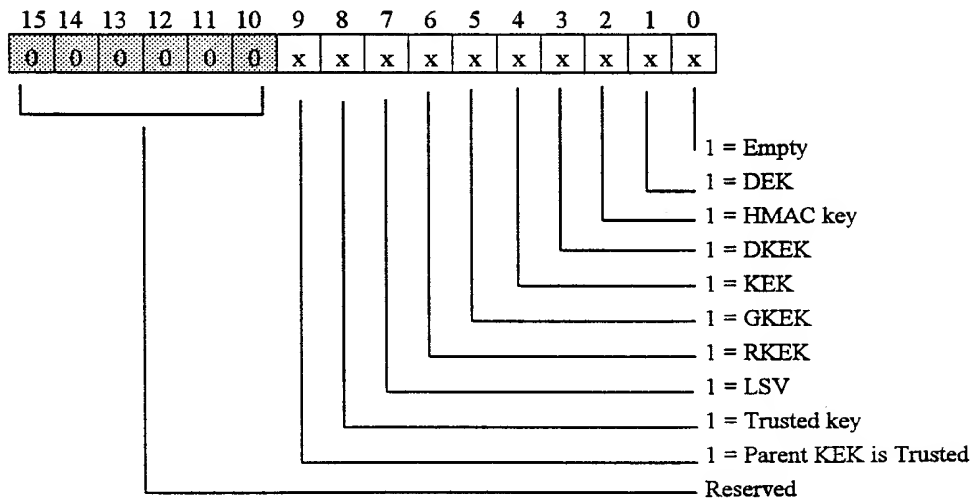
The *type* field specifies the algorithm for which the key is to be used (eg. DES, 3-DES, RC-5, etc.)

5 The *keylength* field indicates the length, in bytes, of the key. For the *CryptIC*, it will be either 8 or 24.

 The *keybits* field is 192-bits long (three 64-bit DES keys). It contains the actual key bits which will be used by the DES or triple-DES algorithm. For single-DES, only 64-bits are used and the remaining 128-bits will be zero. The least
10 significant bit of each byte is considered the DES parity bit and is ignored. The *weakened_keybits* field is 64-bits long and contains the 'original' bit values which were overwritten by 0's during the key weakening process. This is necessary if the key needs to be *Exported* in an interoperable format. Prior to exporting, the key is 'un-weakened' back to its original form. Upon *importing* a key, the weakening process is
15 again performed to restore the original key. This behavior and thus the need to preserve the 'weakened bits' is needed in order to protect against key-size tampering when exporting/importing a key.

 The *keycount* field indicates the length of the key in 64-bit blocks. For the *CryptIC*, it will be either 1 or 3. Although the keycount can be inferred from the
20 keylength field, it is provided in order to optimize performance.

The *attributes* field is a 16-bit value with the following bit definitions, shown in the table below:



This attributes field is created when the key is first generated and stays with the key both internally and when it is off-loaded. The only exception is when a key is *Exported* or *Imported*. In these operations, the key is always considered Untrusted and the import/export operation specifies the key type.

Symmetric Key IRE External Representation

When a symmetric key is off-loaded, the key must be put into an 'IRE-external' form. The external form consists of all of the original random key bits, salt bits, attributes, and SHA-1 message digest. The entire external form is encrypted, including the SHA-1 message digest. The salt bits and attributes are pre-pended to the key bits to provide randomness to skew the external key once it is encrypted (in CBC mode).

[userfield][type][keylength] ...

... $E_{KEK}\{[salt][attributes][key-bits][weakened-bits][keylength][hash-digest][DES-pad]\}$
 ... [32-pad]

Where E_{KEK} indicates the encryption of the data in {} by the specified KEK.

The secretkey object is 576 bits / 72 bytes / 18 dwords long. This contains the following:

Field	Length	Description
Userfield	16	An application-specified field which may be used for any purpose
type	16	Algorithm for which the key was created (DES, triple DES)
keylength	16	Length of the key, in bytes. This field is duplicated inside the covered portion
salt	54	Random bits used to enhance the key encryption process. Similar to an IV
attributes	10	Attribute bitmap. See description above
key-bits	192	The actual DES or triple-DES key bits. If DES, then last 128 bits are zero
weakened-bits	64	The bits overwritten in the key-weakening process
keylength	16	Length of the key, in bytes. Duplicated here to be included in the Hash
hash digest	160	A SHA-1 hash of the key, attributes and salt. Provides an authentication code
DES-pad	16	Zero-pad bits to round-up to the nearest 64-bit boundary (for DES encryption)
32-pad	16	Zero-pad bits to round-up to the nearest 32-bit dword boundary

5 The purpose of the DES-pad field is because DES is 8-byte block oriented and the seven blocks are formed so they can each be properly encrypted without needing padding. The 32-pad field allows the secretkey object to fall on a 32-bit dword boundary which eases handling on the PCI bus.

10 The SHA-1 message digest is computed over the *salt*, *attributes*, *key-bits* and *weakened-bits* fields. Once the message digest is inserted in the object, the *salt* through *message digest* fields are encrypted by a KEK. The encryption mode is typically CBC, although for untrusted keys, the application may specify a different mode. A fixed IV is used for encrypting the key blob, since the random *salt* takes care of randomizing the encryption. This symmetric key blob is then returned to the application.

15 Upon loading-in the Black symmetric key blob, the CGX Kernel decrypts the key blob and runs the SHA-1 hash over the specified bits and compares the message digest to the one stored in the blob. If the message digest comparison fails, the symmetric key is rejected (i.e. not loaded).

Symmetric Key Inter-Operable External Representation

When an application chooses to exchange an IRE symmetric key with another crypto-vendor, the symmetric key must be converted from IRE's storage format into one that is more easily extractable so that it can inter-operate with other crypto-vendors. (Note that keys which are exported from or imported to the *CryptIC* are always considered untrusted.)

To do this, a basic storage format based on Microsoft's CryptoAPI has been adopted. The general form of the symmetric key to be exported/imported is:

$$E_{KEK}\{[salt-bits][key-bits][data-bits]\}$$

Where E_{KEK} indicates the encryption of the data in $\{\}$ by the specified KEK.

The application can specify no salt, it can explicitly define the salt bits, or it can request the *CryptIC* to generate the salt bits. The key bits consist of the Red symmetric key bits as defined in the IRE external symmetric key formation. A DES key will occupy 8 bytes and a triple-DES key (2-key or 3-key) will occupy 24 bytes. The key bits are laid out in big-endian form. The data field is supplied by the application, it can contain anything the application chooses, or it may be omitted if the application desires.

The one to three pieces of external symmetric key (salt, key bits, and data) are put into a buffer that is then encrypted by a symmetric KEK. The attributes and message digest of the IRE External form are not included in this format.

Since the external symmetric key must be covered with a symmetric KEK, the salt bits (if present) must be in multiples of 8 bytes and the data bits (if present) must be in multiples of 8 bytes as well. The exception to this rule is for HMAC keys. In this case the key must be in multiples of 2 bytes. Therefore, an HMAC key with a length of 5 bytes will be exported as 6 bytes with the 6th byte exported as a 0.

In the case of covering the external symmetric key with a public key, the salt bits (if present) can be as many bytes as the application chooses and the data bits (if

present) can be as many bytes as the application chooses. However, the entire size of the salt, key, and data bits must be less than or equal to the size of the public key's modulus used to cover them.

Symmetric Key Weakening Process

5 As shown in Figure 17, the symmetric key weakening algorithm is similar to the one created by IBM called CDMF key shortening. IRE's approach is similar, with the difference in that we write 0s into the key bits starting at 0 going for the amount to be shortened by. In the CDMF approach, various bits are masked with zero.

10 To implement the key weakening scheme, the number of random key bits to be used is important. For 40-bits to 56-bits, 64-bits of random data are to be provided, for 64-bits to 112-bits, 128-bits of random data are to be provided, and for 120-bits to 169-bits, 192-bits of random data are to be provided. The number of random bits specified is critical in that it allows the key shortening algorithm to work correctly.

15 The weakening process consists of two encryptions; thus implying there are at least two fixed symmetric keys used to do this. There are 14 unique fixed symmetric keys used for this process. A set of symmetric keys for each key length between 8-bits and 56-bits is provided; thus 7 pairs of fixed symmetric keys.

20 The algorithm to shortening the symmetric key is first to encrypt a full 64-bits with the first key of the fixed symmetric key set. Once the encryption is complete that output is shortened by overwriting it with zeros for the amount to be shortened by. For example, if a 40-bit key is needed, the first 24-bits of the interim result will be overwritten. Once the zeros are written onto the interim result, it is encrypted again with the second fixed key. The result of the second encryption is then used as the weakened symmetric key. This has the effect of distributing the weakened bits over
25 the entire key.

 Since we allow for key lengths to be between 40-bits and 192-bits with increments of 8-bits we also have to distribute the weakened key between the 3 key

blocks (a block is defined as a 64-bit key block) if the key is to be 64-bits or greater. For 40-bits to 56-bits the key distribution does not occur, it fits in one key block.

To distribute larger keys they are first evenly distributed between the first two key blocks if they can all fit. This means that a 72-bit key is broken up so that 40-bits are in block one and the other 32-bits in block 2. This means that the bits are evenly distributed into blocks 1 and 2 and if the key length is odd (meaning the key length can not evenly distribute into the two key blocks) the extra 8 bits is put into block 1. Then when the key is laid out in the two blocks, block 1 is copied to block 3 to create a triple DES key. However, if there 128-bits or more of key, then the remaining bits are put into block 3 to complete the triple DES key.

HMAC keys are never weakened.

Symmetric Key Access and Protection

Providing access to the symmetric keys has been carefully considered in light of security and flexibility. Since the philosophy of the *CryptIC* is to provide a general-purpose interface, the application is given a variety of importing and exporting rules.

Each of the symmetric key types have their own rules on key access and are addressed in more detail below in their respective sections. Furthermore, depending on the Program Control Data Bits (PCDBs), the application could be allowed to import its own Red keys to be used as KEKs and DEKs.

ASYMMETRIC KEYS

Asymmetric keys allow for the implementation of public key algorithms and digital signatures. The following sections describe the public keysets (public and private) that are used by the three asymmetric cryptographic algorithms supported by the *CryptIC*.

Asymmetric Key Types

Like symmetric keys, the *CryptIC* supports several flavors of asymmetric keys.

The three types are:

Diffie-Hellman public keys:	Used for computation of a negotiated key based on a previously generated modulus and base
RSA public keys:	Used for both encryption and digital signatures
DSA public keys:	Used for digital signatures only

The application is responsible for storage of public keysets, eliminating the need for the CGX Kernel to store public keys. The modulus and public parts of the keyset are stored off-chip in the Red form, while the private portion of the key is covered. When requesting a public key operation, the application is required to provide as a parameter the public keyset it wishes to use. The CGX Kernel, in turn, makes a local copy of the public key and, if necessary, uncovers the private portion into the public key working kernel RAM in the *CryptIC*. When the operation completes, the local copy of the private key is destroyed.

Diffie-Hellman

As in all public key schemes Diffie-Hellman uses a pair of keys, public and private. However, Diffie-Hellman is unique in that it does not perform encryption/decryption or signatures as do the other public key systems. Rather, it implements a means to generate a shared secret or a negotiated DEK (i.e. traffic key or session key).

Unlike RSA, a fixed modulus (n) and generator (g) can be shared among a community of users; therefore, the generation of a new stored modulus will not occur often. The modulus, public key and private keys can be exported/imported from/to the *CryptIC*.

A new public key (X) is generated upon request by the application; many private and public keys can be generated from one common modulus. At the same

time, a private key (x) is also generated. The public key is generated using the newly generated private key ($X = g^x \bmod n$).

5 A shared secret key or DEK can later be created by using the other *CryptIC*'s public key ($DEK = Y^x \bmod n$). The negotiated DEK can be exported from the *CryptIC* but it must first be covered by one of the *CryptIC*'s KEKs (GKEK, or an application-generated KEK).

RSA

10 As in all public key schemes RSA uses a pair of keys, public and private, to encrypt/decrypt and implement digital signatures. However, unlike Diffie-Hellman the modulus can not be fixed. A new modulus must be generated for each new public key pair. The public exponent is calculated, based on a starting point specified by the application (up to 64-bits maximum). Typically, the starting exponent is 65537, which avoids all of the known attacks on the fixed exponent of 3. Therefore, the public key is made up of the modulus ($n = pq$) and an exponent ($e \geq 65537$).

15 The private key (d) is generated by taking the multiplicative inverse of the public key ($d = e^{-1} \bmod ((p-1)(q-1))$). Like the public key, an optimization is performed by using the values of p and q during the decryption and signing operations of RSA. The optimization consists of precomputing d_p and d_q and using the Chinese Remainder theorem (CRT). Therefore, the CGX kernel keeps p and q and the precomputations
20 around as part of the private key.

The key pair can be exported from the *CryptIC*. The public key returned is made up of e and n , the private key returned is made up of d , p , q , d_p , d_q , $p^{-1} \bmod q$, and $q^{-1} \bmod p$. Storing all of these values is essential in order to optimize the RSA decryption process using the Chinese Remainder Theorem. Naturally, allowing these
25 private components to be seen would be a security breach, therefore the private key components are returned covered via one of the *CryptIC*'s GKEKs or an application-generated KEK.

DSA

DSA also uses a pair of keys, public and private, to implement digital signatures only. Unlike RSA, fixed moduli (p and q, and generator g) can be shared among a community of users. Therefore, the generation of a new stored modulus will not occur often. The modulus, public key and private keys can be exported/imported from/to the *CryptIC*.

The public key is composed of two pieces: the public key (y) and the modulus data (p, q, and g). The *CryptIC* will allow a modulus size to be between 512 and 2048 bits with increments of 64 bits.

The private key (x) is a random 160-bit number. For every signature, a new 160-bit k is created. Once the signature is created, k can be destroyed. Furthermore, the signature process creates r and s, they are used in the verification process to generate v which is compared to r to verify a signature. The *CryptIC*'s command CGX_SIGN returns r and s and the command CGX_VERIFY expects r and s as arguments.

A DSA key pair can be exported from the *CryptIC*. The public key returned is made up of y, p, q, and g; the private key returned is made up of x. The private key is returned covered via one of the *CryptIC*'s GKEKs or an application-generated KEK.

Program Control Data Key (PCDK)

The PCDK is a fixed DSA public key that is stored in masked ROM. The PCDK is used to verify the digital signature on a PCDB modification token (this is explained in more detail in the section Programmable Control Data Bits Initialization String (PCDB_IS) or to verify the signature on 'Extended Code' which may be loaded into the chip at run-time. The PCDK is hard-coded into the CGX kernel and cannot be off-loaded or exported.

Asymmetric Key Lengths

Since the *CryptIC* supports several public key algorithms, the requirements for public key pairs and their moduli differ. However, in general the *CryptIC* will support keys and moduli in the range of 512 bits to 2048 bits in multiples of 64 bits.

5 Asymmetric Key Generation

Public key generation like symmetric key generation is very important to the security of the system. Poorly generated public keys or moduli will result in inferior crypto-protection. The main tenet behind public key generation is the generation of random numbers with a high probability of being prime. The *CryptIC* implements a
10 widely accepted method of testing for primality, the Rabin-Miller algorithm. Moreover, all prime numbers generated by the chip will have the most significant bit set and will be odd.

The *CryptIC* will never reuse the same modulus for an RSA or DSA new public key. However, the *CryptIC* does allow for a common exponent to be used for
15 the RSA exponent. The typical choice is to use the a starting point of 65537 as the RSA encrypt exponent.

DSA Key Generation

The application has several configurable ways to generate the public keyset for DSA operations. It can generate the modulus' p and q in a *safe* or *weak* manner. The
20 safe method is based on appendix A of the X9.30 specification. It goes through a formal method of creating p and q from a 160 bit seed and 16 bit counter. In this method the application is ensured of not using a '*cooked modulus*'.

In the weak method, the modulus' p and q are generated in a simple method of finding q a prime factor of p - 1. This doesn't provide for a seed and counter but
25 generates p and q faster.

In either case, p and q are tested to determine if they are prime. As noted earlier, they are tested using the Rabin-Miller primality test and the application can specify the number of times it wants to run the test. The more times it is run, the higher the probability that the number is prime. However, the more times the test is run, the longer it takes to generate p and q. Also, as part of the primality test, the prime number to be tested goes through a small divisor test of the primes between 1 and 257.

The private exponent is generated using the random number generator and the public exponent is derived from the private and modulus data. Furthermore, the application can change the private and public parts as often as it chooses for a given modulus using the CGX_GEN_NEWPUBKEY command.

RSA Key Generation

The RSA moduli, p and q, are generated by first finding a prime number p and then another prime number q that is close in value to p.

In either case, p and q are tested to determine if they are prime. As noted earlier, they are tested using the Rabin-Miller primality test and the application can specify the number of times it wants to run the test. The more times it is run, the higher the probability that the number is prime. However, the more times the test is run, the longer it takes to generate p and q. Also, as part of the primality test, the prime number to be tested goes through a small divisor test of the primes between 1 and 257.

The public exponent is created by finding an e that is relatively prime to $\phi(n)$, the product $(p-1)(q-1)$. The starting point of this search is 65537. In most cases e remains 65537. In the event that $\phi(n) = 65537k$, where $k \geq 1$, then the encryption exponent will be the next largest odd value which is relatively prime to $\phi(n)$.

The private exponent is found by taking the multiplicative inverse of e mod $(p-1)(q-1)$.

Diffie-Hellman Key Generation

The application has several configurable ways to generate the public keyset for Diffie-Hellman operations. It can generate the modulus p in a *safe* or *weak* manner. The safe method finds a *safe prime*, one of $2q + 1$.

5 In the weak method, the modulus p is generated as a random number. In this case, the modulus is weak but will be generated very quickly. Furthermore, the modulus p is not tested for primality.

10 In either case, p and q are tested to determine if they are prime. As noted earlier, they are tested using the Rabin-Miller primality test and the application can specify the number of times it wants to run the test. The more times it is run, the higher the probability that the number is prime. However, the more times the test is run, the longer it takes to generate p and q . Also, as part of the primality test, the prime number to be tested goes through a small divisor test of the primes between 1 and 257.

15 The private exponent is generated using the random number generator and the public exponent is derived from the generator, private exponent and modulus. Furthermore, the application can change the private and public parts as often as it chooses for a given modulus using the CGX_GEN_NEWPUBKEY command.

Asymmetric Key Representation

20 Asymmetric key sets are represented in one of two forms, an *IRE external* form, or an *Inter-operable External* form. The IRE External form is a subset of the Inter-operable External form. The external form is modeled after the Microsoft CryptoAPI specification. Note that the asymmetric key sets contain the Public key, the Modulus/Generator, and the Private key. Only the private key portion is covered by a
25 KEK.

Asymmetric Key IRE External Representation

When an application chooses to create an IRE private key for local storage, this format is used. This is the format that is returned for the CGX_GEN_PUBKEY and CGX_GEN_NEWPUBKEY operations.

5 [userfield][type][keylength][datapage][pointer-to-modulus][pointer-to-pubkey]
[pointer-to-privkey] and:

The modulus/generator element is: [modulus-generator-packed]

The public key element is: [public-keybits-packed]

The private key element is: $E_{KEK}\{[salt-bits][private-keybits-packed]\}$

10 Where E_{KEK} indicates the encryption of the data in $\{\}$ by the specified KEK.

For the private key portion, the application can specify salt, no salt, or request the *CryptIC* to generate the salt bits. Moreover the application can use the salt bits as a means to prepend a formatted packet of data in front of the private key. The key bits are the Red private key bits. For DSA it will just include, x , a 160-bit key. For Diffie-
15 Hellman it will just include the private key, x , a key between 160 and 2048-bits. For RSA the private key bits will be p , q , $d \bmod (p-1)$, $d \bmod (q-1)$, and $q^{-1} \bmod p$, and d (in this order). The key bits are laid out in little-endian form.

The private key bits and salt are put into a buffer that is encrypted by an untrusted symmetric KEK.

20 If salt is to be supplied, it must be in 16-bit units, no single bytes allowed. Furthermore, the total byte count of salt, and key bits must be a multiple of 8-bytes in order to fall on a DES block boundary.

Asymmetric Key Inter-Operable External Representation

25 When an application chooses to exchange an IRE private key with another crypto-vendor, the private key must be translated from IRE's storage format into one that is more easily extractable.

To do this, a basic storage format based on Microsoft's CryptoAPI has been adopted. The general form of the private key element to be exported/imported is:

The private key element is: $E_{KEK}\{[\text{salt-bits}][\text{key-bits}][\text{data-bits}]\}$

Where E_{KEK} indicates the encryption of the data in $\{\}$ by the specified KEK.

5 The application can specify salt, no salt, or request the *CryptIC* to generate the salt bits. Moreover the application can use the salt bits as a means to prepend a formatted packet of data in front of the private key. The key bits are the Red private key bits. For DSA it will just include, x , a 160 bit key. For Diffie-Hellman it will just include the private key, x , a key between 160 and 2048 bits. For RSA the private key
10 bits will be p , q , $d \bmod (p-1)$, $d \bmod (q-1)$, and $q^{-1} \bmod p$, and d (in this order). The key bits are laid out in little endian form. The data field is supplied by the application, it can contain anything the application chooses or it can be left out if the application does not require it.

15 The private key (salt, key bits, and data) are put into a buffer that is encrypted by an untrusted symmetric key.

If salt or data are to be supplied, they must be in 16 bit units, no single bytes allowed. Furthermore, the total byte count of salt, key bits, and data bits must be a multiple of 8 bytes in order to fall on a DES block boundary.

20 KEY HANDLING REQUIREMENTS

An important requirement to defining a key management scheme is setting forth requirements on how the keys are handled by the CGX Kernel and application. The CGX Kernel presents several requirements on key handling for both public and symmetric keys. This is followed by a detailed discussion of secret/public key
25 hierarchy and control.

The following are requirements the application must abide by in order to create and manipulate keys using the *CryptIC*:

- 1) By default, Red key exchange between the application and the CGX Kernel is only allowed in the direction from the application to the CGX Kernel (i.e., loading), for all user symmetric keys (KEKs and DEKs). Absolutely no Red key exportation is allowed.
- 5 2) All keys stored in the internal KCRs are in the Red form, no internal Black key storage is allowed.
- 3) Black keys are not allowed as part of encryption or public key operations. In other words a key must be uncovered, in its Red form, before it can be used in any of the CGX Kernel commands. This is true for the private key of a public
10 keyset as well.
- 4) All key management commands supported by the *CryptIC* (e.g. CGX_GEN_KEY, CGX_GEN_KEK, or CGX_GEN_PUBKEY, etc.) are atomic. This means that when a key is created, derived, or negotiated, a covered (i.e. Black or encrypted) copy of the key is returned at the same time,
15 as part of the same command. Therefore, the only way to get back a key (i.e. the Black one) is via the command that created it. This prevents an intruder application from hijacking a key.
- 5) All keys used or stored by the *CryptIC* fall under two umbrellas of protection, trusted or untrusted. Trusted keys can never be exposed in their Red form; the
20 *CryptIC* stores them in a secure and protected manner. An untrusted key can be used in many flexible ways but secure protection is left to the application.
- 6) All keys in a trusted tree (see Figure 18) can not move from that tree, they are permanently stored in the tree.
- 7) All keys in an untrusted tree (see Figure 18) can move to other untrusted trees
25 and/or a trusted tree, once under a trusted tree the key can not move (as stated in item 6 above).

- 5 8) There are two classes of keys, IRE external keys (symmetric and public) and interoperable keys (symmetric and public). IRE external keys contain secure attributes that describe its trust and key type. Interoperable keys contain nothing but the key material, formatted with optional salt bits in the front and optional application-supplied data bits at the end of the key material.
- 9) Internal IRE keys contain two attributes: Use and Trust level. The use attributes specify the key type (ie. KEK, DEK, DKEK). The trust level is Trusted or Untrusted (as explained in item 5 above). The use attribute mandates how a key can be used and which commands will allow it.
- 10 10) As a means to combat many types of cipher-text attacks on Red key storage, all symmetric keys and private keys (of a public keyset) will include 'Salt' which are random bits prepended to the key material prior to encrypting. The random salt data helps to prevent statistical cipher-text attacks; thus better securing a covered key.
- 15 11) In order to off load and protect user keys securely (i.e. Diffie-Hellman negotiated ones, CGX_GEN_KEY ones, or imported Red user keys) a generated internal key encryption key (GKEK) must be used. This provides an internally protected conceptual boundary around the *CryptIC* device and the application's non-volatile storage memory. This is because the GKEK is an
- 20 extension of the *CryptIC*, it is never exposed and the keys covered under it are therefore never exposed.
- 12) The LSV is laser programmed at the factory and is unique for each *CryptIC* device.
- 13) The LSV is a two-key, triple-DES key (i.e. 112 bits), all GKEKs are covered
- 25 under the LSV using triple DES in the CBC mode.
- 14) All GKEKs are internally generated using the CGX_GEN_KEY command.

- 15) To off-load a GKEK, it must be covered under the LSV (no other covering key is permitted).
- 16) All GKEKs are triple DES keys (i.e. 192 bits), all user keys are covered under the GKEK using triple DES in the CBC mode. The CGX Kernel supplies a fixed IV.
- 17) The LSV can not be exported in any form. Furthermore, the LSV can only be used by the CGX_GEN_KEY, CGX_GEN_RKEK or CGX_UNCOVER_KEY commands.
- 18) GKEKs can not be exported/imported, although they may be off-loaded in IRE External form. Furthermore, GKEKs can only be used to cover user keys, not as traffic keys. Therefore, the GKEK can only be used by the CGX_UNCOVER_KEY, CGX_GEN_KEY, CGX_DERIVE_KEY, CGX_LOAD_KEY, CGX_GEN_PUBKEY, and CGX_GEN_NEWPUBKEY commands.
- 19) A user key can be created internally via CGX_GEN_KEY (for symmetric keys) and CGX_GEN_PUBKEY (for public keys), imported in the Red form, derived via CGX_DERIVE_KEY, or negotiated via the Diffie-Hellman key exchange (symmetric key only).
- 20) User keys are also known as KEKs, DEKs (a symmetric key), or a Public keyset. All user keys must be covered by a GKEK, RKEK or KEK in order to be exported.
- 21) Once a user key is covered under a GKEK it can not be covered by any other key. In other words, the user key can not be propagated down or across in the symmetric key hierarchy. This rule prevents key stealing or spoofing.
- 22) User keys covered by a KEK can use either the DES or triple-DES algorithm. For untrusted keys, the application is responsible for providing an IV for the appropriate modes (i.e., CBC, CFB, and OFB).

23) The depth of the symmetric key hierarchy is unlimited, KEKs can be covered under KEKs 'N' deep as long as it abides by these key handling rules. This is the same for trusted KEKs as well.

24) The CGX Kernel knows if a KCR contains a key, the key type, the key length, and its attributes. Anything beyond that is the responsibility of the application.

Symmetric Key Handling

The *CryptIC* provides support for eight kinds of symmetric keys: LSV, hardware only, protection, generated, loaded, imported, negotiated, transformed, and derived. The master key is the unique LSV laser programmed into each *CryptIC* device. The LSV provides the ability for the application to safely off-load key material with out the possibility of exposing the key's Red contents.

Hardware only keys, DKEKs, are internally generated symmetric keys (via the CGX_GEN_KEY operation) that are used to cover symmetric keys, DEKs, only. The DKEK keys allow applications to load Black symmetric DEKs into the hash/encrypt crypto interface to be uncovered and used as traffic keys. This allows the application to use the hash/encrypt hardware interface securely since keys are never exposed in the Red form.

Protection keys, GKEKs, are internally generated symmetric keys (via the CGX_GEN_KEY operation) that can only be covered by the LSV. The protected keys allow the application to build a symmetric key tree hierarchy that to the application is protected internally within the *CryptIC* device. This protection is a conceptual boundary around the *CryptIC* and the application's non-volatile memory storage. Protected keys are returned in the Black form.

Generated keys are user keys generated internally to the *CryptIC*, they never reside in the Red form outside the bounds of the *CryptIC* device. All generated keys are created via the CGX_GEN_KEY operation. An internal user key can be used as a

KEK or DEK, but is always known as a trusted key by the CGX Kernel unless the application requests it to be untrusted.

5 Loaded keys are user keys loaded in the Red form. They can be used as a KEK or DEK, but is always known as a untrusted key by the CGX Kernel. The loaded key may be covered by a GKEK or another trusted or untrusted KEK. Loaded keys play an important role for applications such as ANSI X9.17 where the application must supply the master keys. The master keys are then used to cover the session keys that can be generated by the *CryptIC* or imported.

10 Imported keys are user keys imported from another vendor's crypto-system, in the Black form. They can be used as a KEK or DEK, but is always known as an untrusted key by the CGX Kernel. The imported key can be covered by a GKEK or a trusted/untrusted KEK.

15 Negotiated secret keys are the result of a Diffie-Hellman key exchange. Negotiated user keys can be used as KEKs or Ks, but are always known as an untrusted key by the CGX Kernel. The negotiated key can be covered by a GKEK or a trusted/untrusted KEK.

20 Transformed symmetric keys are the result transforming an HMAC key into a useable DES, Triple DES, or RC5 key. The transforms are all based on the current IPsec working group papers on HMACs. Transformed user keys can be used as KEKs or DEKs, but are always known as an untrusted key by the CGX Kernel. The negotiated key can be covered by a GKEK or a trusted/untrusted KEK.

25 Derived keys are the result of a cryptographic operation that converts a passphrase into a symmetric key using a one-way HASH algorithm. Derived keys can be used as KEKs or DEK, but are always known as an untrusted key by the CGX Kernel. The derived key can be covered by a GKEK or a trusted/untrusted KEK.

Figure 18 presents the symmetric key tree hierarchy supported by the commands of the *CryptIC*.

The keys shown in Figure 18 represent their type (in upper case: LSV, GKEK, KEK, DKEK, or DEK), trust level (the lower case letter to the left of key type, t for trusted and u for untrusted), generation method (the superscript as defined below), and the key movement method (the subscript as defined below).

5 !e = not exportable

e = exportable

g = generated

n = negotiated (Diffie-Hellman)

10 l = loaded

i = imported

t = transformed

d = derived

15 Since the *CryptIC* does not contain non-volatile storage, the CGX Kernel must allow a means for the application to safely off-load symmetric keys into some external non-volatile storage for future use. This is accomplished through the classification of symmetric keys shown in Figure 18 above. All trusted keys (the ones in the white oval) are securely protected via the LSV symmetric key hierarchy. All the other keys (in the gray shaded area of the box) are untrusted and the storage scheme is left to the application.

20 The trusted tree provides a means to securely store symmetric keys. The storage is secure because the parent KEK is a trusted key and by definition a trusted key means that the key was created in a manner that prevents it from being exposed in the Red form. Therefore, the child keys under a trusted KEK are securely protected because the parent KEK is not compromised.

25 A trusted key must contain attributes. The attributes are used to define the class of the key and its type. The class defines the key as a trusted or untrusted and the type defines the key to be used as a KEK or data key (DEK) only; it can't be both.

For all IRE symmetric keys, the type and class will be securely stored with the key material. Therefore when a key is reloaded into the *CryptIC*, the type and class is reliably restored.

At the root of the trusted tree resides the LSV. The diagram shows that the LSV is trusted (tLSV), it has been laser programmed (LSV^{laser}), and is not exportable (LSV_{ie}). The LSV is the master KEK for the *CryptIC*; it is used as the covering KEK of protected keys, GKEKs; it can not be used by the general encryption commands.

Under the LSV resides the protection keys, the GKEKs. The sole purpose of these keys is to limit the exposure of the LSV so that the application can not determine the LSV from plain-text attacks. The GKEKs are trusted keys (tGKEK), are internally generated so that the Red key material is not easily detectable (GKEK^s), and are not exportable (GKEK_{ie}).

Under the GKEKs, the application is free to build its own trusted tree or make it a flat one which resides directly under the GKEK and no deeper. To build a deeper secure trusted tree, the application must create trusted KEKs. The trusted KEKs (tKEK) shown in Figure 18 are formed via the CGX_GEN_KEY command (KEK^s) and can never be exported (KEK_{ie}). A trusted KEK is similar to the GKEK in all aspects except that its parent KEK is either a GKEK or another trusted KEK; not the LSV. Therefore, the trusted tree can be as deep as one wants but the parent of each branch must be a trusted KEK. Furthermore, all child keys under a trusted KEK can not be exported, they are permanently stored under the trusted KEK.

All session keys (DEKs) stored under a trusted KEK or GKEK can be either trusted or untrusted. The trust state of these keys does not matter; it's the fact that they are stored under a trusted parent that matters. Being a trusted DEK has one slight difference then an untrusted DEK. The difference is that trusted DEKs (tDEK) shown in Figure 18 are formed via the CGX_GEN_KEY command (DEK^s). Untrusted DEKs are created in some untrusted means therefore the Red key material

could be determined more easily. However, like trusted DEKs untrusted DEKs under a trusted KEK can not be exported.

Before describing the untrusted trees there is one more interesting point about trusted trees. In Figure 18, there are two untrusted KEKs which both reside under trusted KEKs (one under a tKEK and the other under a tGKEK). They are both
5 untrusted and therefore their children reside in the untrusted umbrella (the gray area). Although the untrusted KEKs can not be exported (because their parent KEKs are trusted) their children can because their parent is untrusted.

Untrusted trees are ones where the parent KEK of the tree is untrusted.
10 Therefore, all keys regardless of depth under the initial untrusted parent will be marked as untrusted and will be allowed to be exported. No trusted DEK or KEK can be created under an untrusted KEK.

A good example of the untrusted tree is shown in the bottom right side of Figure 18. The parent KEK is untrusted (i.e. KEK) and could have been created in a
15 multitude of ways ($\text{KEK}^{\text{enlid}}$, generated, negotiated, loaded, imported, and derived) and is exportable (KEK_e). All of its children are marked as untrusted and could have been created in many ways, as with its parent. The child keys are exportable.

There is another type of an untrusted tree, called the dangling tree. All
dangling trees are by definition are untrusted because the root key is always an
20 untrusted KEK or it has no root KEK at all (ie. an untrusted DEK). The upper right hand of Figure 18 shows an example of a dangling tree. In this example the parent is the root and is an untrusted KEK. Furthermore, the parent KEK is not exportable because there is no parent KEK to cover it.

One example of this dangling untrusted tree is the case where the untrusted
25 parent or root KEK was created as a derived key (i.e. via a pass-phrase). The application then stores keys under this derived untrusted KEK. It can now safely store away the untrusted keys and not have to store the parent KEK. Anytime it wants to

use one of the stored keys it must use the CGX_DERIVE_KEY operation to regenerate the untrusted parent KEK.

Another example of a dangling tree is shown in the lower left corner of Figure 18. In this example the dangling tree is flat, it is a single untrusted DEK. This key is untrusted and can not be exported because it has no parent to uncover it with. However, the real usefulness of this key is its use as a session key for one time only.

The DKEK keys are not shown in the diagram in Figure 18 because they function identically to the KEKs in the diagram with one exception. The exception is that the only keys that can be covered under them are DEKs of any type (i.e. derived, loaded, etc). At no time will a KEK or another DKEK be allowed under a DKEK. The DKEK key is reserved for the use of the hardware crypto-block interface. To allow applications to load in Black symmetric DEKs from an external context store (ie. in PCI or External memory space) and not have to expose their keys in the Red form.

Asymmetric Key Handling

Asymmetric keys contain three components: the public exponent, the modulus and generator, and the private exponent. The private exponent can be imported in the Red form, but only exported in the Black form (the private key must be covered). Public key sets can be internally generated using the CGX_GEN_PUBKEY or CGX_GEN_NEWPUBKEY (generates the exponents of a Diffie-Hellman or DSA public keyset) commands.

All asymmetric keys are known as untrusted keys, therefore they can be used to cover symmetric keys to be exported or be exported themselves.

Public keys can be used as covering keys, key exchange keys, and signature keys. However, the application should create different keys for these tasks to protect the keys from various attacks. The *CryptIC* does not enforce key types for public keysets as it does with symmetric keys.

A public keyset can only be covered by a trusted or untrusted symmetric key, not another public key. Only the private part of the public keyset gets covered; the public key and modulus/generator are always in the Red form.

5 **SOFTWARE DATA OBJECTS**

This section describes the various software data objects that make up the CGX Kernel.

Configuration Data

10 The CGX Kernel maintains the program control data bits. The functionality of the bits are described in the section Kernel Configuration String (KCS). The following sub-sections describe the data storage definitions for the program control data bits.

Program Control Data Bits

The PCDB allows the application to customize the CGX Kernel. For example, 15 the PCDB contains a bit to allow the exportation of Red key material. However, the application must obtain an unlock-data message to get the proper permission to permanently change the non-volatile copy of the PCDBs. Without the appropriate unlock-data message the application is denied access to the PCDBs. Obtaining the unlock-data message and the process of using it is discussed in the section Kernel 20 Configuration String (KCS).

Figure 19 specifies the data definition for the PCDB bits.

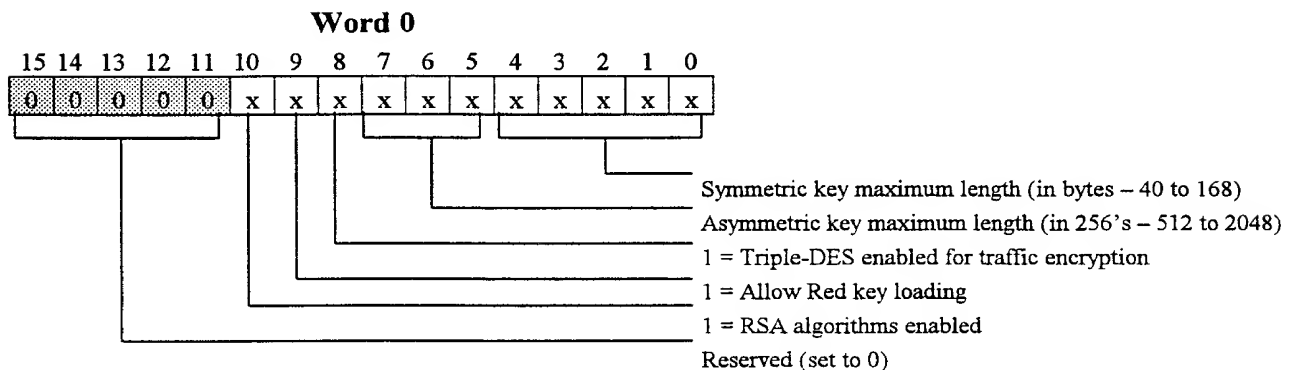
PCDB Bit Definitions

The following lists the PCDB words in lowest memory order, defining the bits that are programmable by the application:

25 **Word 0:**

As shown in the table below:

- Bits 0 - 4 set the maximum allowed symmetric key (DES, triple-DES) length. The starting size is 40-bits and these bits allow a step size of 1 byte up to 168-bits.
- Bits 5 - 7 set the maximum allowed asymmetric key (RSA, DSA, D-H) length. The starting size is 512-bits and these bits allow a step size of 256-bits up to 2048 bits.
- Bit 8 selects whether triple-DES is allowed for traffic encryption. It is always allowed for Key encryption.
- Bit 9 selects whether the *CryptIC* will allow Red key loading.
- Bit 10 selects whether the RSA public key algorithms are enabled. A royalty fee is due to RSA for those chips which have this feature enabled.
- Bits 11 - 15 are reserved.

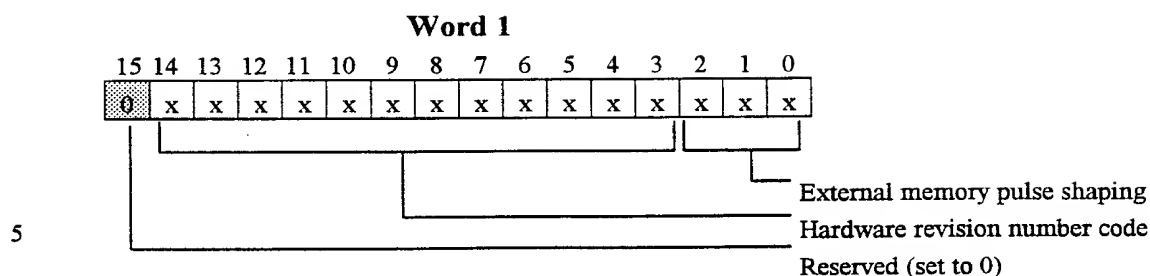


Word 1:

As shown in the table below:

- Bits 0 - 2 set a hardware delay-line which is used for adjusting the timing pulse for External Memory access. This is typically configured at the factory to account for IC process changes.
- Bits 3 - 14 set the hardware revision number of the *CryptIC*. This value is returned in the CGX_GET_CHIPINFO command.

- Bit 15 is reserved.



Command Interface

The major component to the command interface is the use of the shared memory, the *kernel block*, to allow command communication between the application and the CGX Kernel. The following sub-sections defined the *kernel block* and its two members: the *command block* and the *status block*.

Kernel Block

The *kernel block* is the CGX Kernel's pipeline or socket to the outside world (i.e. the application). The *kernel block* provides the application and CGX Kernel with the ability to communicate with one another. The *kernel block* is a pre-formatted shared block of RAM located in the DSP's data address space, either in internal DM or external DM. The *kernel block* is shown in Figure 20.

The *kernel block* is comprised of two areas: the *command block* and the *status block*. The *command block* is used as the data transfer area between the application and CGX Kernel for servicing cryptographic commands. The *status block* is used to keep the application updated on the status of the CGX Kernel.

The *kernel block* consists of a status member and a pointer to the *command block*. Allowing the blocks to be defined as pointers allows the application to build static *kernel blocks* with the possibility of globalizing the *status block*. By globalizing the *status block* the application can reference the single instance; thus making status checks easier to perform. The section Kernel Block Object defines the *kernel block* as a very simple C data structure.

Moreover, the shared block approach is important for two other reasons. The first is to make the software interface to the CGX Kernel compiler-independent.

Otherwise, the interface would be achieved via arguments pushed onto the software stack. All compiler languages support different calling frames; therefore, it would be difficult to come up with a simple scheme to handle all languages easily and efficiently.

The second reason for the shared block approach is to optimize the access to the CGX Kernel. Using the shared block approach, the application can set up static *kernel blocks*. A static *kernel block* would have pre-configured arguments and commands. The application would reuse the *command block* and never have to update it, similar to using a DMA controller. For example, the application could set up a static traffic encryption *command block*. The *command block* would be set up once with the following arguments: CGX_ENCRYPT command, the crypto-block context object, a data-in and data-out pointer, and a data-in size. Then for every block(s) of data to encrypt the application would only invoke, `cgx_secure_transfer()`, with a pointer to the static *kernel block*. Like a DMA operation, the input buffer is obtained from the data-in pointer and the output buffer is obtained from the data-out pointer. The application would only have to replenish the data-in buffer and consume the data in the data-out buffer after the service completes. This would save processing time because the application would only have to populate the *kernel block* once prior to traffic.

The status field provides the CGX Kernel status back to the application (ie. return result code). The status field is a place holder for cryptographic service result codes, the acceptable status codes are specified in the section Status Definitions.

Kernel Block Object

The *kernel block* object is made up of two independently addressable blocks. Figure 21 illustrates the kernel block.

The *kernel block* uses a pointer to the *command block* to allow for global substitutions of the block. For example, the application may have many static

command blocks for several cryptographic services while it only has one *kernel block* for the entire application.

The 4 word kernelblock object may reside in either the internal DM data page or in any external DM page of the *CryptIC*. It is important however that when the CGX call is made (transfer vector), that the DMOVLAY register is set to the page in which the kernelblock object resides. The CGX Kernel will not modify the DMOVLAY register until it has read-in the *kernel block*.

Member Details

The DeviceNo member, as shown below, is used to allow the application to specify which class of command is being requested and the memory model for the *command block* arguments.

DeviceNo Code	Operation	Memory Model
0x0000	ByteCode Command	DSP Internal PM
0x0001	ByteCode Command	DSP Internal DM
0x0002	CGX Command	PCI (32-bit addresses)
0x0003	CGX Command	DSP DM (14-bit address, data page)
0x0004	CGX Command	Mixed Mode (dp = bitmap)

DeviceNo codes 0 and 1 are reserved for special ByteCode commands which are used in factory testing. Codes 2 – 4 are used for CGX commands.

DeviceNo code 0x0002 specifies that if any of the 10 arguments in the command block are pointers, then they will be defined as 32-bit PCI address pointers. For example, if for a given CGX command, argument #3 is a pointer to a data source, then it will be a 32-bit PCI address.

DeviceNo code 0x0003 specifies that if any of the 10 arguments in the command block are pointers, then they will be defined as DSP Data Memory (DM) address pointers. The least-significant word of each argument will hold the 14-bit DM address. The most-significant word for each argument will hold the 16-bit DMOVLAY value for the DM address, thus allowing the pointer to reference any page in internal or external *CryptIC* memory.

DeviceNo code 0x0004 specifies that the dp member of the kernel block is a bitmap, specifying for each of the 10 arguments whether they are a pointer into PCI memory space (1) or an address/overlay page in DSP data memory space (0). Again, this only applies for those arguments which are pointers rather than explicit data. For example, a CGX command could be executed with a *command block* pointing to data values both in local DSP memory as well as PCI memory. This avoids having to copy data into the DSP memory prior to executing a CGX command.

The member, dp, is provided to allow flexible location of the *command block* within the CryptIC's memory spaces. The *kernel block* (which includes the *status block*) will be in whatever page is currently active (specified by the DMOVLAY register) when CGX was entered. However, the dp member allows the application to allocate the *command block* in any of the external data pages available. This avoids having to use the valuable internal data space if the application chooses not to. As part of the CGX Kernel entry operation the CGX Kernel will save the previous data page and set the 2183 data page to the one passed in the *kernel block*. Upon exit the CGX Kernel will restore the old data page it saved in the entry operation.

The application may use the status field to determine if a cryptographic service completed successfully or not. The status code will indicate whether the CGX Kernel is busy or not, and will provide a success or failure result code. Note: the application must never modify the status field. The CGX Kernel is the only software that should write to the status field. If the application modifies the status field it may not know if the CGX Kernel is active.

Command Block

The *command block* provides the argument interface between the application and the CGX Kernel. The *command block* is comprised of several fields as shown in the C data structure definition below. The *command block* is made up of a command

field and argument list. The command field is loaded with the operation code. The available commands are listed in the section Command Definitions.

The argument list is used to pass-in the command arguments. The argument list is a fixed array of void pointers. The void pointer allows you to establish a pointer to any object as well as passing in arguments by value.

Although most of the commands take a variable number of arguments, the *command block* is always fixed with 10 argument locations. However, the application does not have to assign the unused arguments to anything if the command does not use them. See the section Command Specification & Arguments for a detail listing of command argument specifications.

Command Block Object

The *command block* object is used by the application as the communication channel to the CGX Kernel or as the argument list for a cryptographic service. The *command block* is made up of a command (i.e. the micro commands as defined in the section Command Definitions), and the list of arguments (a maximum of 10 arguments are allowed). The argument list can contain pointers to various objects to be passed into the CGX Kernel. Figure 22 illustrates the *command block* object.

KEY OBJECTS

Secret Key

As shown in Figure 23, the secret key object, *secretkey*, is used to represent a symmetrical key between the application and the CGX Kernel. The data structure contains the key length (i.e. length, allowing the maximum key length of 192 bits for the triple DES algorithm), application specific data, the secret key type (i.e. DES, triple DES, etc.), and the raw buffer (i.e. *k*[16]).

The raw buffer, *k*, houses the actual secret key. As part of the secret key, some salt bits are pre-pended to the buffer, *k*. There are a total of 54 bits of salt

added, followed by 10 bits of key attributes. The key attributes (for details see the section Symmetric Key IRE Internal Representation) define the type of key (i.e. KKEK, KEK, DEK etc.) and its trust level (i.e. Trusted or Untrusted). The next 192 bits contain the actual key bits, followed by 64 bits of overwritten key_weakening data. Then, the 16-bit keylength field appears, followed by a 160-bit SHA-1 digest of the first 256 bits of k. The digest is used to authenticate the salt, attributes, and key.

The secretkey object is only used as a means to communicate and securely store keys off of the chip. Internally, the chip uses the same secretkey structure minus the last 192 bits of the buffer, k, the SHA-1 digest data.

All secret keys will have a type defined as: CGX_DES_A or CGX_TRIPLE_DES_A. The secret key type is used by the CGX Kernel so that it can internally transform the secret key to be usable for the secret key algorithm specified in the type field.

The secret key is stored in a buffer of 16 bit units or WORD16 data types. This allows for more efficient storage and access schemes. By default, the key is stored with its most significant bits in the upper bits of the final WORD16 item. The salt data will fill-in the unused least significant bits of the raw key buffer.

The extra field is 16 bits of data provided to the application for what ever it chooses. The CGX Kernel does not read or write this field. Furthermore, it will preserve this field while the key is internally stored in volatile or non-volatile memory.

Public Key

The public key object, publickey, is used to represent an asymmetric key that is to be used between the application and the CGX Kernel. The data structure is made up of a modulus, public key, and private key objects. The publickey object can accommodate Diffie-Hellman, DSA, and RSA public keys; otherwise known as the public keysets. The publickey object can only manage a maximum key length and/or

modulus length of 2048 bits. Public key modulus length can be regulated with PCDBs. If required, an 'Enabler Token' can override the factory laser settings of the PCDBs.

5 Figure 24 defines a generic public keyset. The type field identifies the kind of public key contained in the structure. Based on the type field, the application assigns pointers to the appropriate keyset objects: modulus, privkey, and pubkey objects for the appropriate type (i.e. Diffie-Hellman, RSA, and DSA). These pointers reference blocks of data stored in little-endian form.

10 All public keysets are stored in a packed form. Packed keys are defined as key structures in which the least significant byte of the next public key structure member abuts the most significant byte of the current member. In this way, fragmentation within data structures is minimized and portability of data is enhanced.

15 The length field specifies the length (in bits) of the modulus for this public key object. The extra field is 16 bits of data provided to the application for whatever it chooses. The CGX Kernel does not read or write this field. Furthermore, it will preserve this field while the public keyset is internally stored in volatile or non-volatile memory.

The following sections define the public keyset objects for each of the algorithm types supported.

20 ***Diffie-Hellman Public Keyset***

The Diffie-Hellman keyset implements the following formula: $Y = g^y \bmod n$. The two variables g and n are combined to form the modulus, Y is the public key, and y is the private key. The contents of the private key must be protected. Therefore, the variable y of the DHprivkey object must be covered before it can be exported. Also, a salt field is included to avoid cipher-text attacks on the covered Diffie-Hellman private key. Figure 25 illustrates the three objects that make up a Diffie-Hellman keyset.

25

RSA Public Keyset

The RSA keyset implements the following formulas: $c = m^e \bmod n$, $m = c^d \bmod n$, where $n = p \cdot q$. The variable n is the modulus, e is the public key, and p , q , and d make up the private key. The variables p , q , $d \bmod (p-1)$, $d \bmod (q-1)$, and $q^{-1} \bmod p$ are kept around as for optimization; they are used by the Chinese remainder theorem. This data must be protected as the private key is. Therefore, the privkey object must be covered before it can be exported. Also, a salt field has been added to avoid cipher-text attacks on the covered RSA private key. Figure 26 illustrates the three objects that make up a RSA keyset.

DSA Public Keyset

The DSA keyset's variables implements the following signing formula: $r = (g^k \bmod p) \bmod q$, and $s = (k^{-1} (H(m) + x \cdot r)) \bmod q$. The following variables implement the verification formula: $u1 = (H(m) (s^{-1} \bmod q)) \bmod q$, and $u2 = (r (s^{-1} \bmod q)) \bmod q$, and $v = ((g^{u1} y^{u2}) \bmod p) \bmod q$, where $y = g^x \bmod p$. The variables p , q , and g make up the modulus, y is the public key, and x is the private key. The other variables r , s , and v are results from the actual calculation and are not stored in a DSA public keyset. However, the contents of the private key must be protected. Therefore, the variable x of the DSAPrivkey object must be covered before it be exported. Also, a salt field has been added to avoid cipher-text attacks on the covered DSA private key. Figure 27 illustrates the three objects that make up a DSA keyset.

Digital Signature

The digital signature object, signblock, is used to represent a digital signature for DSA public key only. The data structure contains the sign result vectors.

The DSA signblock is represented as two vectors, r and s , to house the results of the DSA digital signature sign operation; stored in two buffers of 16 bit units or WORD16 data types. This allows for more efficient storage and access schemes. By

default, the two vectors are stored with its most significant bits in the upper bits of the WORD16 buffers. Figure 28 shows the DSA signblock object definition.

Seed Key

5 For generating reproducible prime numbers for DSA modulus', p and q, a seed key object is required. The seed key object is used by the application to provide a 160 bit seed value for generating the primes and for a counter to be used in verifying the primes.

The DSA seedkey object is represented as a 32 bit counter and a 160 bit seed buffer. Figure 29 shows the DSA seedkey object definition.

10 **Key Cache Register Data Type**

Key cache registers are represented by a register ID value. The register ID can be between 0 and 14 (the maximum allowable key cache registers). However, key cache register number 0 is reserved for reference to the LSV (laser-trimmed) key cache register. The key cache register data type is shown in Figure 30.

15 **Red Keys**

The CGX Kernel allows the importation of Red key material, both secret and public keys.

Context Management

20 Context management for the CGX Kernel is a critical requirement from the point of view of security, real-time, and storage. Context management allows the application to interleave several occurrences of a cryptographic operation. For example, the CGX Kernel only has one KG (or crypto-block) to encrypt with. However, some applications (e.g. packet networks) may have several encryption
25 sessions going on at the same time. If the CGX Kernel did not implement context management then it could not start an encryption session for another party until the

current party released the resource. This is because the use of the crypto-block requires state information about the encryption session to be saved between calls to the encryption command, CGX_ENCRYPT. If the state information isn't saved between calls then the next block will not encrypt correctly. So without context management,
5 the application cannot share various resources of the CGX Kernel.

Context Stores

Like the *kernel block* the context stores are defined as blocks of memory with pointers to buffers specified by byte counts or implicit byte counts; thus providing a language (i.e. compiler or assembler) independent block interface.

10 *Symmetrical Encryption Context Store*

The context store for symmetrical (i.e. secret key) cryptographic commands (CGX_ENCRYPT and CGX_DECRYPT) are shown in Figure 31.

The size of the crypto_cntxt block is 12 bytes of contiguous memory. The config field is used to select the encryption configuration (i.e. modes, loading
15 configuration, and feed-back count); the constants to use for this field are defined in the document titled, *CryptIC Command Interface Specification*. The size of the KCR ID is 2 bytes. Note that the iv buffer size is fixed to 8 bytes. However, depending on algorithms used in the future, the iv buffer can grow easily if need be. This is because the iv buffer is at the end of the object and can be added to and still allow for backward
20 compatibility.

The state information to restore the secret key context is saved in the buffer, iv, after each call to the CGX_ENCRYPT or CGX_DECRYPT commands. The application is responsible for priming the iv buffer with an initial random value when starting a new encryption session for a key. Moreover, to implement a simple
25 resynchronization service, the application only needs to modify the iv buffer. The CGX Kernel writes back the feed-back register of the hardware crypto-block into the iv buffer for the CBC, CFB, and OFB modes after each call to CGX_ENCRYPT or CGX_DECRYPT. The iv buffer must remain unchanged by the application to

maintain synchronization with the receiving end of the encryption session. There is no iv buffer required in the case of the ECB mode; the iv buffer pointer is ignored and the application is not required to set it. Another advantage of the crypto_cntxt is that the application has full access to the iv (i.e. the feed-back register or crypto-block state information) at any time.

One-Way HASH Context Store

The context store for the one-way HASH cryptographic commands (CGX_INIT_HASH and CGX_HASH_DATA) is shown in Figure 32.

The size of the hash_cntxt block is 96 bytes of contiguous memory. The algorithm field is used to select the one-way HASH function algorithm (i.e., MD5 or SHS); the constants to use for this field are defined in the document titled, *CryptIC Command Interface Specification*. Note that the digest buffer size is variable depending on the algorithm used; MD5 uses a 128-bit digest and SHS uses a 160-bit digest. Only after closing the hash will the message digest field point to the message digest. The length of the state information is fixed for all algorithms based on the use of a union. The length of the state buffer is defined to be 96 bytes or 48 words.

SOFTWARE INTERFACES

This section describes the two major software interfaces, the command interface and the CGX overlay.

Command Interface

As discussed earlier the command interface provides the API between the CGX Kernel and the application. This is accomplished via a shared memory block and a transfer vector. The shared memory is implemented via the *kernel block* and the transfer mechanism is via specialized hardware logic.

Shared Memory

The *kernel block* (defined in the section Kernel Block) provides the shared memory necessary for the CGX Kernel and application to communicate with one another. Furthermore, unlike other shared memory schemes the *kernel block* is not a static area. The *kernel block* is reestablished for each cryptographic command request. This has the advantage of allowing the application to dynamically allocate the *kernel block* on the fly or better yet controlling which pieces of the *kernel block* (the command and/or *status blocks*) are statically versus dynamically allocated. The advantage with this is that the application can setup several static copies for each of the cryptographic commands before hand. Thus reducing the overhead of having to repopulate the command fields before each call to the CGX Kernel.

Like everything there are disadvantages to not having a fixed static *kernel block*. The most interesting disadvantage is that the CGX Kernel becomes a synchronously driven engine; not asynchronous. This is because if it was asynchronous, the CGX Kernel could post events, asynchronously, to the *status block* to report the health of its crypto-functions in real-time. However, the CGX Kernel does not run asynchronously; it only runs when it is invoked by the application and the rule is that it has control of the processor until it completes. Furthermore, the CGX Kernel will only modify the current *kernel block* while a cryptographic command is executed.

CGX Kernel Transfer Vector

In order to execute a cryptographic service, the application must explicitly invoke the CGX Kernel. To do this a special transfer vector mechanism has been created. The transfer mechanism is a specific call to address 0x2000 instruction with the PMOVLAY register set to 0x000F. The AR register must contain a pointer to the *kernel block* which defines the command requested.

When the software call instruction occurs, the special hardware logic (along with some hardware signals and counters) overlay the ROM encoded CGX Kernel

program block over a portion of the application's program space. Once control has been passed to the Kernel, the application is prevented from monitoring the CGX Kernel's ROM, RAM or register areas.

5 To avoid forcing the application to understand the transfer mechanism, a single transfer vector routine is supplied by IRE to be used by the application. Furthermore, all of the cryptographic command definitions are provided as 'wrap code' around this single transfer operation; thus alleviating the application of supporting this operation.

CGX Kernel Transfer Vector Operation

10 The CGX Kernel transfer operation is fast and simple. The general idea behind the transfer operation is to establish a calling frame (*kernel block* and *command block*), transfer into the command mode of the CGX Kernel, execute the command, and return back to the application with the results.

15 The CGX Kernel transfer operation may be implemented as a function call by the application. A header file defining the operations for all of the cryptographic services is provided. These definitions consist of a macro that is used as wrap code to the transfer vector operation. The macros populate the *kernel block* with the appropriate commands and arguments. For example, to encrypt a block of data using the encryption command, CGX_ENCRYPT, the application would use the macro wrap definition illustrated in Figure 33 and found in the cgx.h header file provided by IRE to the customer. The cgx.h header file is provided to the application as the
20 software interface to the CGX Kernel cryptographic services. The application only needs to use this definition file and the transfer operation, _cgx_transfer_secure_kernel, to interface to the CGX Kernel.

25 As can be seen in Figure 33, the wrap code prepares the *kernel block* for communications between the application and the CGX Kernel. (Optimizations to the wrap code have been previously suggested in the description of the kernel block. It is possible that the application can establish the kernel block once and just call the transfer operation cgx_transfer_secure_kernel over and over. This would speed things

up so that the kernel block populate code would not be re-executed. This would only be useful for repetitive commands, like traffic encryption.) Looking at the example closer, one notices a call to the operation, `cgx_transfer_secure_kernel`, at the end of the macro wrap code `cgx_encrypt`. The transfer operation,

5 `cgx_transfer_secure_kernel`, performs the actual transfer of control from the application to the CGX Kernel. The transfer operation is responsible for assigning the `kernelblock` object pointer to the AR register and forcing a software interrupt to cause control to be transferred to the CGX Kernel.

When the transfer operation is invoked it simply performs these four steps:

- 10
- First it saves the return address to the calling function (i.e. the application) on the software stack,
 - second it populates a general purpose register with the pointer to the *kernel block* in the AR register,
 - third it transfers control to the CGX Kernel by forcing a software interrupt, and
 - 15 • fourth the CGX Kernel returns back into the transfer operation in the application memory space, because of the PC stack save caused by the software interrupt, and it returns to the calling application's operation because of the saved address in step 1.

20 CGX Overlay Interface

As discussed earlier, the main purpose of the overlay operation is to provide the proper wrap code around the CryptoLIB operations. This allows the total reuse of the CryptoLIB software and hiding the *CryptIC* platform specific hardware functions from the application and the CryptoLIB software. To do this, there is a one to one

25 mapping of CGX overlay operations (the wrap code) to each cryptographic command supported by the CGX Kernel.

CGX Overlay Table Definition

Figure 34 shows a single CGX overlay tuple. The CGX overlay table contains N CGX overlay tuples or entries. The member *cgxf* is a pointer to the CGX overlay operation that is invoked by the CGX Kernel if the control variable satisfies the necessary requirements. The control variable contains a preemption bit.

CGX Overlay Table Processing

To simplify things, the CGX overlay operations reside in a table of CGX overlay tuples as defined by the data structure, *cgx_overlay_tuple*, that is indexed by the *cmd* field of the *command block* of the *kernel block* (e.g. *kb->cb->cmd*). Once the CGX Kernel determines which tuple out of the CGX overlay table is the correct tuple to access, it must perform another check to determine if the application has proper access to the tuple.

To do this, the CGX Kernel must first verify the preemption bit (part of the control field) allows it to enter the command mode. The preemption bit occupies the most significant bit (i.e. bit 15) of the control variable; thus the preemption mask value is 0x8000. If the bit is a 1, it must enter the *command_block* mode. In the *command_block* mode, the CGX Kernel will not let another command preempt the current command. If the bit is a 0, the CGX Kernel will enter the command mode where the current command can be preempted by a secret key encryption/decryption or one-way Hash command.

COMMAND INTERFACE RESOURCES

Besides the set of CGX cryptographic commands, the application is presented with various resources, mainly in the area of configuration registers and storage. As in most processor platforms, the developer is presented with a set of instructions and

memory stores (either memory or registers) to manipulate data; with the CryptIC the application is also presented with memory storage capabilities.

INITIALIZATION PROCESSING

5 The initialization process of the CGX Kernel can be invoked at any time via the CGX Command Interface. To invoke the initialization processing of the CGX Kernel, the CGX_INIT command is issued. This command allows the application to customize the CGX Kernel in two ways: the Kernel Configuration String and PCDB settings.

10 To do this, the CGX Kernel allows the application to pass in initialization strings for the two areas. The initialization strings are an array of defined bits to allow the application to set, disable, and enable various features of the CGX Kernel. The following sub-sections define the initialization strings allowed as part of the CGX_INIT command.

15 Kernel Configuration String (KCS)

 The Kernel Configuration String is a set of bit-mapped words which control certain features in the CGX kernel. These features include: Semaphore handling and feature enables. The KCS object definition is shown in Figure 35.

Programmable Control Data Bits Initialization String (PCDB_IS)

20 Program Control Data Bits (PCDBs) are a set of bit-mapped words which control certain features in the CryptIC. These features include: Symmetric and Public key lengths, RED key load enable, Algorithm enables, etc. The CryptIC is laser-programmed at the factory with a default set of PCDBs.

 The PCDB_IS allows the application to customize the CGX Kernel PCDBs.
25 For example, the PCDB_IS allows the application to give the CGX Kernel permission to change lengths of key material. However, the PCDB_IS needs an *unlock-data*

message to gain permission to change the PCDBs. This message is digitally signed using an IRE private key, and the CryptIC verifies the signature with its public key. Without the *unlock-data* message the application is denied access to the PCDBs and the PCDB_IS is ignored.

5 The PCDB_IS's first word is used to specify the number of words succeeding the first word. If the application only wants to program data in the first word of the PCDB_IS it would only need to set the size word (the first word of the PCDB_IS) to a 0x1, and then pass in a PCDB_IS buffer of two words. The remaining words take on the factory default settings. Furthermore, depending on the unlock-data message, the
10 application may not gain permission to all of the PCDBs. The CGX Kernel will ignore any bits in the PCDB_IS that the application has no permission to change in the PCDBs.

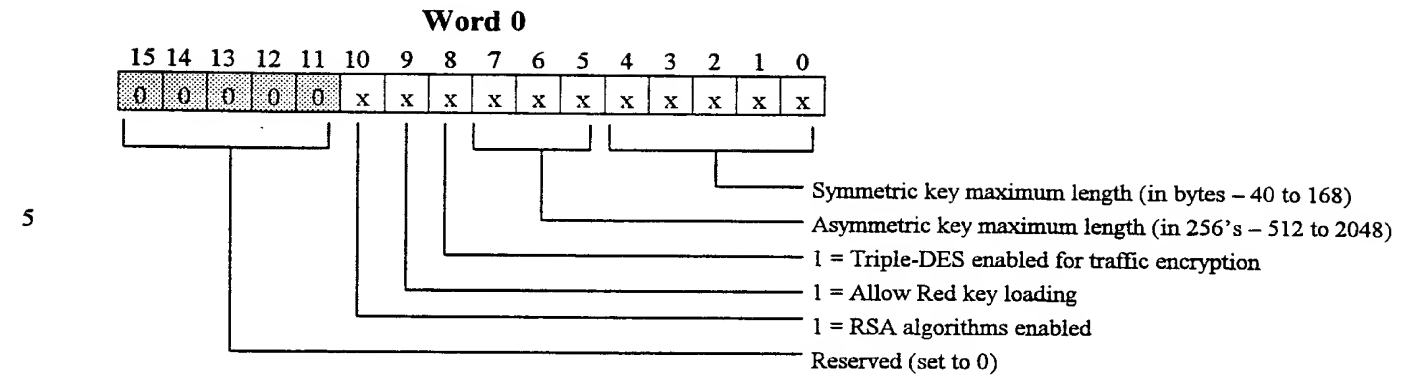
 The following lists the PCDB_IS words in lowest memory order, defining the bits that are programmable to the application:

15 **Word 0:**

 As shown in the table below:

- Bits 0 - 4 set the maximum allowed symmetric key (DES, triple-DES) length. The starting size is 40-bits and these bits allow a step size of 1 byte up to 168-bits.
- Bits 5 - 7 set the maximum allowed asymmetric key (RSA, DSA, D-H) length.
20 The starting size is 512-bits and these bits allow a step size of 256-bits up to 2048 bits.
- Bit 8 selects whether triple-DES is allowed for traffic encryption. It is always allowed for Key encryption.
- Bit 9 selects whether the *CryptIC* will allow Red key loading.
- 25 • Bit 10 selects whether the RSA public key algorithms are enabled. A royalty fee is due to RSA for those chips which have this feature enabled.

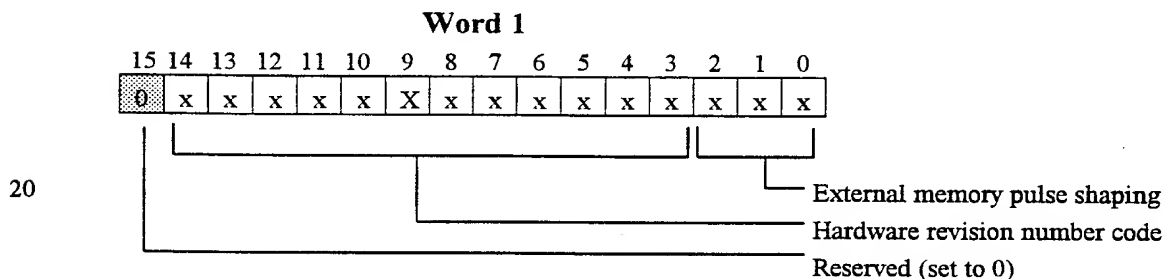
- Bits 11 – 15 are reserved.



10 Word 1:

As shown in the table below:

- Bits 0 - 2 set a hardware delay-line which is used for adjusting the timing pulse for External Memory access. This is typically configured at the factory to account for IC process changes.
- 15 • Bits 3 - 14 set the hardware revision number of the *CryptIC*. This value is returned in the CGX_GET_CHIPINFO command.
- Bit 15 is reserved.



KEY STORAGE

- 25 The CGX Kernel provides several areas for storage of symmetric and public keys. It contains three areas: Key RAM, Key Cache Registers (KCRs), and the Application Space. The Key RAM is a working area for one public key. A working area implies that it is volatile; in other words the public key is lost on power-down and

resets. All public key operations (i.e., encryption or digital signatures) use the volatile key RAM for that single public key storage and for intermediate results.

5 The volatile Key Cache Registers provide a working area for many symmetric keys. Again, this is a working area so all the keys will be lost during a power-down or reset. The 15 volatile Key Cache registers are addressable via register Ids from 0 through 14 (where 14 is the last addressable register). The key cache register, 0, is reserved for the LSV symmetric key. The command interface uses the key cache register Ids as the addressing mechanism for storage, retrieval, and encryption.

10 Note: Additional key storage can be achieved within the CryptIC through the use of extended RAM. A parameter can be passed in the CGX_INIT command to request allocation of increments of 1kbyte of internal DSP Data RAM to key storage. Up to 700 symmetric keys can be accomodated in this manner.

15 The Application Space can be used for non-volatile or long time storage for persistent key material. The CryptIC does not contain non-volatile memory (other than the Laser-programmed bits). However, the section Key Management discusses how keys can be stored off chip securely using the commands discussed in this memo.

COMMAND INTERFACE DEFINITIONS

20 The following sub-sections provide a listing of the command interface definitions. The definitions provide the proper argument values for micro commands, status information, and object definitions. These definitions must be used in order to communicate correctly with the CGX Kernel.

COMMAND DEFINITIONS

25 The following table represents the valid command definitions:

Name	Value
General Commands	
CGX INIT	0x0000
CGX DEFAULT	0x0001
CGX RANDOM	0x0002
CGX GET CHIPINFO	0x0003
Encryption Commands	
CGX UNCOVER KEY	0x0004
CGX GEN KEK	0x0005
CGX GEN KEY	0x0006
CGX LOAD KEY	0x0007
CGX DERIVE KEY	0x0008
CGX TRANSFORM KEY	0x0009
CGX EXPORT KEY	0x000A
CGX IMPORT KEY	0x000B
CGX DESTROY KEY	0x000C
CGX LOAD KG	0x000D
CGX ENCRYPT	0x000E
CGX DECRYPT	0x000F
Public Key Commands	
CGX GEN PUBKEY	0x0010
CGX GEN NEWPUBKEY	0x0011
CGX GEN NEGKEY	0x0012
CGX PUBKEY ENCRYPT	0x0013
CGX PUBKEY DECRYPT	0x0014
CGX EXPORT PUBKEY	0x0015
CGX IMPORT PUBKEY	0x0016
Digital Signature Commands	
CGX SIGN	0x0017
CGX VERIFY	0x0018
Extended Algorithm Commands	
CGX LOAD EXTENDED	0x0019
CGX EXEC EXTENDED	0x001A
Hash Commands	
CGX HASH INT	0x001C
CGX HASH DATA	0x001D
CGX HASH ENCRYPT	0x001E
CGX HASH DECRYPT	0x001F
Math Commands	
CGX MATH	0x0021

Name	Value
PRF Commands	
CGX PRF KEY	0x0020
CGX PRF DATA	0x001F
CGX MERGE KEY	0x0021
CGX MERGE LONG KEY	0x0022
CGX LONG KEY EXTRACT	0x0023
RKEK Commands	
CGX GEN RKEK	0x0024
CGX SAVE KEY	0x0025

Table 13 Command Definitions

STATUS DEFINITIONS

The following table represents the allowable status definitions, returned by the CGX Kernel:

Command	Value	Description
CGX_SUCCESS_S	0x0000	Cryptographic service completed successfully.
CGX_WEAK_KEK_S	0x0001	A weakly generated KEK (a secret key) was used as an argument. The KEK is rejected and the service fails.
CGX_EMPTY_REG_S	0x0002	An empty key cache register was referenced, the service fails.
CGX_INVALID_REG_S	0x0003	An invalid key cache register was given, the service fails.
CGX_ACCESS_DENIED_S	0x0004	Attempt to export RED key material but the RED_KEY_EXPORT_ENABLED PCDB bit is not enabled, the service fails.
CGX_WEAK_KEY_S	0x0005	A weakly generated key (a secret key) was used as an argument. The key is rejected and the service fails.
CGX_HARDWARE_FAILURE_S	0x0006	One or more hardware devices used to implement the requested service has failed.
CGX_BAD_MODULUS_S	0x0007	A bad modulus (i.e., not greater than 0) was used.
CGX_INVALID_KEY_LEN_S	0x0008	Invalid secret key length used. Only 40, 64, and 128 bit length secret key lengths allowed.
CGX_BAD_MODE_S	0x0009	Invalid mode (i.e., only ECB, CBC, CFB, and OFB modes allowed) used. The service fails.
CGX_PRIVILEGE_DENIED_S	0x000A	Invalid login ID and/or PIN, or login ID does not have privilege access to cryptographic operation.
CGX_INVALID_CMD_S	0x000B	Invalid command opcode (i.e., constant val.) provided.
CGX_INVALID_LEN_S	0x000C	Invalid length (i.e., constant value) provided. Returned for public key operations only.
CGX_BAD_KEYSET_S	0x000D	Specified public keyset source location is empty or contains the incorrect public keyset.
CGX_INVALID_SIGNATURE_S	0x000E	The digital signature verification failed; the signature of the application's message does not match the signature of the verification signature block.
CGX_BIGNUM_FAIL_S	0x000F	Public key processing failed due to large integer failure (i.e., overflow condition).
CGX_BAD_KEK_S	0x001F	Invalid KEK or NULL when KEK argument required.
CGX_RAM_FAIL_S	0x0011	low level RAM diag's failed
CGX_ROM_FAIL_S	0x0012	low level ROM diag's failed
CGX_LSV_FAIL_S	0x0013	low level LSV CRC test failed
CGX_NULL_PTR_S	0x0020	NULL Pointer passed when argument required.
CGX_FAILED_TOKEN_SN_S	0x0028	cmd failed token serial number check
CGX_FAILED_TOKEN_VERIFY_S	0x0029	cmd failed token verify signature
CGX_FAILED_RKEK_S	0x002A	cmd failed RKEK
CGX_FAILED_NEG_KEY_S	0x002B	failed Diffie Hellman negotiate key
CGX_FAILED_SAVE_KEY_S	0x002C	failed save key operation
CGX_FAILED_GEN_RKEK_S	0x002D	failed gen RKEK operation
CGX_FAILED_LOAD_KEY_S	0x002E	failed load key operation
CGX_PROG_NOT_RUN	0x002F	Start address 0 supplied (to IRE internal test

Command		Value	Description
			programs)
CGX_STARTED	S	0x00FC	Application has made the request to CGX.
CGX_RUNNING	S	0x00FD	CGX has accepted the call and is running.
CGX_FAIL	S	0x00FE	General catch-all failure code.
CGX_BUSY	S	0x00FF	Secure Kernel Busy, can not be preempted.

Table 14 *Status Definitions*

ALGORITHM, MODE, AND KEY USAGE DEFINITIONS

5 This section defines the algorithms, mode, and key usage definitions for symmetric key, public key, digital signatures, and one-way Hash operations.

Key Usage Definitions

10 The following table defines the key usage bits that must be specified when generating, loading, negotiating, or deriving a public or symmetric key. The constant values referenced in the table must be used in order to use an IRE symmetric or public key within the confines of the CryptIC device.

Name	Value	Description
CGX_KCR_DEK	0x0002	This key definition is reserved for symmetric keys that are to be used for traffic or message encryption only.
CGX_KCR_DKEK	0x0008	This key definition is reserved for symmetric keys that are to be used to cover other symmetric or public keys. The covered key must be a DEK, not a KEK.
CGX_KCR_KEK	0x0010	This key definition is reserved for public keys and symmetric keys that are to be used to cover other symmetric or public keys.
CGX_KCR_UNTRUSTED	0x0100	This constant can be or'ed with any of the three definitions above to force a newly generated, loaded, negotiated, or derived symmetric or public key to be known as an untrusted key. This is important if the application intends to export a particular key; only untrusted keys can be exported. Furthermore, the untrusted key must not be under (i.e. covered by) a trusted parent or it can't be exported as well.

Table 15 Key Usage Definitions

Symmetric Key Algorithms

5 The following tables define the symmetric key algorithms and their operating modes. The constant values referenced in the tables must be used in order to access the CGX Kernel's symmetric key algorithms.

Name	Value	State Context Length (i.e., the dynamic IV)	Description
DES			
CGX_DES_A	0x0001	0 bits (0 bytes)	Single DES: ECB, CBC, CFB, and OFB
Triple DES			
CGX_TRIPLE_DES_A	0x0002	128 bits (16 bytes)	Triple DES: ECB, CBC, CFB, and OFB
RC5			
CGX_RC5_A	0x0004	0 bits (0 bytes)	Single RC5: ECB, and CBC
HMAC			
CGX_HMAC_A	0x0008	0 bits (0 bytes)	Only used by the CGX_TRANSFORM_KEY command for IPsec key transforms.

Table 16 Symmetric Key Algorithm Bits

Name	Value	Description
CGX ECB M	0x0001	Electronic CodeBook Mode. No IV
CGX CFB M	0x0002	Cipher FeedBack Mode. IV required
CGX OFB M	0x0004	Output FeedBack Mode. IV required
CGX CBC M	0x0008	Cipher Block Chaining Mode. IV required
CGX_1BIT_FB	0x0010	Feedback bits for both the OFB mode and the CFB modes above. Defines 1-bit feedback.
CGX_8BIT_FB	0x0020	Defines 8-bit feedback.
CGX_64BIT_FB	0x0040	Defines 64-bit feedback.

Table 17 *Symmetric Key Mode Bits*

5 The symmetric key algorithm bits are supplied in the symmetric key object (discussed in the memo, *CryptIC Software Architecture*), the key type field. When symmetric keys are created or imported into the CryptIC the type of the symmetric key specified in the key type field of the symmetric key object defines the type of symmetric key algorithm to use. The symmetric key algorithm bits are not used as part of the algorithm field of the crypto_cntxt object.

10 Also, the bits (7 through 9) of the algorithm word, of the algorithm field of the crypto_cntxt object, is used to contain the special encryption control bits. Currently, there are two bits used to specify how keys are loaded into the KG during encryption. The masks are *or'ed* into the algorithm and mode constants defined above.

Name	Bit Mask	Description
CGX_AUTOLOAD_C	0x0080	This mask is used to request the CGX Kernel to perform an auto load key when encrypting. The use of auto load means that if the key is already loaded into the KG (i.e., the crypto-block) don't repeat the key load for this operation. This is the default mode.
CGX_FORCELOAD_C	0x0100	This mask is used to request that the CGX Kernel always loads the key when encrypting.
CGX_NOLOAD_C	0x0200	This mask is used to request that the CGX Kernel perform no key load when encrypting.
CGX_RESYNC_C	0x0400	This mask is used to request that the CGX Kernel re-start the OFB or CFB mode. To restart traffic synchronization it only has to encrypt the IV before it can encrypt/decrypt traffic.

Table 18 *Symmetric Key Load Definitions*

Public Key Algorithms

5 The following table defines the supported public key algorithms (this includes digital signatures as well) and their operating modes. The constant values referenced in the table must be used to access the CGX Kernel's public key algorithms.

Name	Value	Description
CGX_RSA_A	0x0001	RSA public key, encryption and signatures.
CGX_DSA_A	0x0002	DSA public key, signatures.
CGX_DH_A	0x0003	Diffie-Hellman public key, key exchange.

Table 19 *Public Key Algorithm Definitions*

One-Way Hash Algorithms

10 The following table defines the supported one way HASH algorithms and their operating modes. The constant values referenced in the table must be used to access the CGX Kernel's one-way HASH algorithms.

Name	Value	Msg. Digest Length	Description
CGX_MD5_A	0x0000	128 bits (16 bytes)	Specifies the MD5 one way HASH algorithm.
CGX_SHS_A	0x0001	160 bits (20 bytes)	Specifies NIST's one way HASH algorithm. SHS-1.

Table 20 *One Way HASH Algorithm Definitions*

IPsec Symmetric Key Transformation Algorithms

5 The following table defines the supported IPsec symmetric key transformation algorithms supported. The constant values referenced in the table must be used to access the CGX Kernel's CGX_TRANSFORM_KEY command.

Name	Value	Description
CGX_XOR_V	0x0001	This operation exclusive-ors a data pattern with the symmetric key and then HASHes it.
CGX_PREPEND_V	0x0002	This operation pre-pends the data pattern to the symmetric key and then HASHes it.
CGX_APPEND_V	0x0004	This operation appends the data pattern to the symmetric key and then HASHes it.

Table 21 *IPsec Symmetric Key Transformation Algorithm Definitions*

COMMAND SPECIFICATION & ARGUMENTS

The following portion of this description defines the command set by specifying in detail a description of the command, the argument list specification (i.e., the *kernel block* and *command block* configuration), and result codes. When an asterisk is pre-

5 pended to the command name it implies the command is optional and it may not be implemented.

General Commands

INIT (Initialize Secure Kernel)

10 **Command Name:** CGX_INIT

Command Description:

15 The Init command is used to initialize the secure Kernel; the Init command can be used at any time. However, the entire command interface, volatile key RAM, and volatile key cache registers are reset; the processor is not reset. The Init command allows the application to customize the secure Kernel's PCDBs, the number of Key Cache Registers (from 15 to 700) and the command interface using the KCS. In particular the application must present the unlock-data message at this time to reprogram any PCDBs that it has permission to change.

20 The command takes, as an input, pointers to unsigned character buffers. The use of unsigned char is to standardize on a little endian memory model as the default. Using the Init command, the application can then change the endian memory model to the memory model it desires.

25 There are three programmable areas: Program Control Data Bits via the pcdb initialization string (includes the signed token), extending the Key Cache Registers by requesting to lock internal DM,

and the Kernel Configuration via the kc initialization string. See the section Software Data Objects for detailed information defining the PCDB and KCS initialization strings.

Command Interface:

```

5      /* initialize the secure kernel */
      cgx_init( kernelblock *kb,
                unsigned char *pcdb,
                unsigned char *kdat,
10     unsigned char *kstart,
                unsigned char *kc)

      Arguments:
          kb->cb->cmd = CGX_INIT;

15     /* the PCDB initialization string */
          /* set to NULL if no initialization required */
          /* the first byte must contain string length in bytes */
          kb->cb->argument[0] = (VPTR)pcdb;
          /* extended key RAM pointer */
20     /* set to NULL if no extended KCR required */
          kb->cb->argument[1] = (VPTR)kdat;
          /* extended key RAM start pointer */
          /* set to NULL if no extended KCR required */
          kb->cb->argument[2] = (VPTR)kstart;
25     /* the Kernel Configuration String */
          /* set to NULL if no initialization required */
          /* the first byte must contain string length in bytes */
          kb->cb->argument[3] = (VPTR)kc;

30

```

Status:

```
kb->sb->status = CGX_SUCCESS_S
```

DEFAULT (Restore Factory Default Settings)

```
35      Command Name:      CGX_DEFAULT
```

Command Description:

The Default command is used to undo all of the initialization processing that occurred via the CGX_INIT command. This command

forces the secure Kernel to reset all of the settings (i.e., command interface and PCDBs) back to factory default settings.

The Default command can be used at any time. However, the entire command interface, volatile key RAM, volatile key cache registers, and PCDB settings are reset; the processor is not reset.

Command Interface:

```
/* Restore factory default settings */
cgx_default( kernelblock *kb )
```

Arguments:

```
kb->cb->cmd = CGX_DEFAULT;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S
```

RANDOM (Generate Random Numbers)

Command Name: CGX_RANDOM

Command Description:

The Random command is used to obtain random bytes of data. The random bytes come from the random number generator hardware device. The application is allowed to request between 1 and 65535 bytes of random data at a time.

Command Interface:

```
/* Generate a random number */
cgx_random( kernelblock *kb,
            VPTR      rbuf,
            unsigned short rbuf_size )
```

Arguments:

```
kb->cb->cmd = CGX_RANDOM;
```

```
/* the number of random bytes, between 1 and 65535 bytes */
kb->cb->argument[0] = (VPTR)rbuf_size;
/* a buffer to hold the requested random numbers */
```

```
kb->cb->argument[1] = (VPTR)rbuf;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S, or
5 CGX_HARDWARE_FAILURE /* RNG hardware failure */
```

See Also: CGX_TEST

GET CHIPINFO (Return CryptIC System Information)

10 **Command Name:** CGX_GET_CHIPINFO

Command Description:

The Get_Chipinfo command is used to obtain system information about the secure Kernel. The system information is returned in a data structure defined as:

```
15 typedef struct _chipinfo {
    WORD16 hw_vsn; /* contains the hardware version number */
    WORD16 sw_vsn; /* contains the software version number */
    WORD16 kcr_max; /* returns max number of KCRs */
    WORD16 kcr_used; /* returns bit-map of active KCRs */
    20 WORD16 pcd[N]; /* contains all of the program control */
    /* data bit values */
    WORD16 serial_number[10]; /* unique CryptIC ID */
} chipinfo;
```

25 The application need not use the data structure; however it must pass a pointer to a buffer of the same size as the chipinfo data structure. In any case, the returned data reflects the current status of the secure Kernel's programmable values and its hardware and software release numbers.

30

Command Interface:

```
/* obtain information about the secure Kernel and CryptIC */
cgx_chipinfo( kernelblock *kb,
              VPTR      cbuf )
```

35

Arguments:

```
kb->cb->cmd = CGX_GET_CHIPINFO;
```

```

/* a buffer to hold the requested chip info, */
/* must allow enough space as defined in data structure above */
/* in fact an instance of the data struct can be used */
kb->cb->argument[1] = (VPTR)cbuf;

```

5

Status:

```
kb->sb->status = CGX_SUCCESS_S
```

See Also: CGX_INIT

10

Encryption Commands

UNCOVER KEY (Load And Decrypt A Secret Key)

Command Name: CGX_UNCOVER_KEY

15

Command Description:

The Uncover Key command allows a generated KEK (i.e. GKEK) or user created key to be decrypted and stored in the key cache. The generated or internal GKEK can only be uncovered by the LSV (i.e. KCR 0) and the user keys can be uncovered by a generated or internal KEK or a user key (i.e. KKEK or KEK), not the LSV. The Uncover Key operation allow user keys to be uncovered using any of the supported secret key algorithms (i.e., DES, Triple DES, etc.), in any of the modes (i.e., ECB, CFB, OFB, and CBC). As for GKEKs, they can only be uncovered by the LSV in the triple DES CBC mode; any attempt to bypass this will fail. The operation copies the secret key from the application's memory in the covered form (i.e., BLACK) and decrypts it into the source key cache register, key.

20

25

30

Like the rule for uncovering the GKEK, any key to be uncovered by a GKEK must also use the triple DES CBC mode. Also, an IV will be returned in the application supplied IV buffer of the crypto_cntxt object. Therefore, the basic requirement is that the parent KEK (i.e. LSV) to GKEKs must be a triple DES key and the GKEK

itself must be a triple DES key. This implies the use of triple DES in the CBC mode when these two KEKs are used for uncovering and covering of keys.

Prior to invoking the encryption command, the application must setup the crypto_cntxt block. The application is responsible for setting the mode, masking in one of the special key load options (described below), setting the appropriate secret key KCR ID, and for populating the iv buffer. If the key is a traffic/encryption key (i.e., keys that are not GKEKs), the IV buffer must be populated with the same IV used by the cover operation. Therefore, the application must have archived the IV buffer along with the covered key.

The iv buffer of the crypto_cntxt block must be read- and write-able; the iv buffer is used to maintain the feedback register for the CBC, CFB, and OFB modes. In the ECB mode the iv buffer is ignored. However, in the case of uncovering keys, the IV buffer is not updated as is done in normal encryption. This is because the original IV may be needed to uncover the key again later.

The Uncover Key command allows the application to mask in one of the special configuration and mode control bits defined in section 0, into the algorithm field of the crypto_cntxt block. The control bits are used by the secure Kernel to determine how to load secret keys before the actual encryption of the plain text takes place. The control bits allow the application to request one of these options: auto-load, force-load, or no-load. Auto-load allows the secure Kernel to check which key is currently loaded into the KG, if it's the same as the key specified for the encryption command it is not loaded; otherwise the key is loaded. Force-load tells the secure Kernel to always load the key. No-load tells the secure Kernel to not load the key; more than

likely the application has already loaded the key (maybe via the CGX_LOAD_KG command). By default, auto-load is assumed.

Command Interfaces:

```

5      /* single encryption mode for key uncover */
      cgx_uncover( kernelblock *kb,
                  kcr      key,
                  secretkey *black_key,
                  crypto_cntxt *cc)

```

Arguments:

```

10      kb->cb->cmd      = CGX_UNCOVER_KEY;

      /* uncovered and store in key cache register number, key */
      kb->cb->argument[0] = (VPTR)key;
15      /* the secretkey object holding key to uncover */
      kb->cb->argument[1] = (VPTR)black_key;
      /* configuration and mode definition, KEK kcr, crypto_cntxt */
      kb->cb->argument[2] = (VPTR)cc;

```

Status:

```

20      kb->sb->status = CGX_SUCCESS_S,
                  CGX_EMPTY_REG_S,
25      CGX_BAD_MODE_S, or // GKEK only uses 3DES, CBC
                  CGX_INVALID_REG_S

```

See Also: CGX_COVER_KEY

30 GEN KEK (Generate An Internal Key Encryption Key)

Command Name: CGX_GEN_KEK

Command Description:

35 The Generate KEK command allows the application to generate an internal key encryption key (i.e. a GKEK) to be used to cover user secret keys or public keys. The GKEK is generated using the output of the random number generator. In order to return the GKEK it must be covered with the LSV, no other secret key is allowed.

Besides generating a GKEK, this command places a copy of the key in its RED form in a KCR location and returns a BLACK copy of the key all at one time; thus it is an intrinsic key function. By returning a BLACK copy of the GKEK immediately, another application is prevented from obtaining the GKEK before the application responsible for its creation does so. Furthermore, this avoids the potential of duplicate copies of a GKEK; thus strengthening its trust.

The GKEK is viewed as a trusted KEK by the secure Kernel. Therefore, it can never be exported from the device in a BLACK or RED form after it has been created and the initial BLACK copy was returned to the application.

The newly generated KEK is left in the specified key cache register, destkey, in the RED form.

The Generate KEK command only generates Triple-DES secret keys. Once a GKEK is created, the secure Kernel only recognizes the GKEK as an internal key. Furthermore, the newly generated GKEK can only be used to cover/uncover application secret keys and public keys. Moreover, when an uncover operation is invoked to load a GKEK, the application is not allowed to specify any algorithm; the secure Kernel assumes triple DES in CBC mode. Also, the application is not allowed to provide a IV; the secure Kernel provides it. This holds true for keys that are covered under a GKEK as well.

Command Interfaces:

```
/*generate a secret key GKEK */
cgx_gen_kek( kernelblock  *kb,
              kcr          destkey,
              secretkey    *bk)
```

Arguments:

```
kb->cb->cmd      = CGX_GEN_KEK;
```

```
/* KCR ID number to place newly generated GKEK */
kb->cb->argument[0] = (VPTR)destkey;
/* BLACK secretkey to return the covered GKEK in */
/* The GKEK is covered by the LSV */
kb->cb->argument[1] = (VPTR)bk;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S,
                CGX_FAIL_S,
                CGX_INVALID_REG_S, or
                CGX_HARDWARE_FAILURE_S /* RNG hardware failure */
```

See Also: CGX_UNCOVER_KEY

GEN RKEK (Generate a Recovery Key Encryption Key)

Command Name: CGX_GEN_RKEK

Command Description:

An RKEK is a Diffie-Hellman negotiated, triple DES, trusted symmetrical key. The RKEK is created by an application, an escrow agent and IRE safenet trusted services to produce the two parts of a negotiated key. IRE safenet trusted services delivers to the application a signed token that contains the chip's serial number and the public key necessary to create the negotiated key. The RKEK can only be generated once the token has been verified.

Besides generating an RKEK, this command places a copy of the key in its RED form in a KCR location and returns a BLACK copy of the RKEK covered by the LSV all at one time; thus it is an intrinsic key function. By returning a BLACK copy of the RKEK immediately, another application is prevented from obtaining the RKEK before the application responsible for its creation does so. Furthermore, this avoids the potential of duplicate copies of a RKEK; thus strengthening its trust.

The RKEK is viewed as a trusted KEK by the secure Kernel. Therefore, it can never be exported from the device in a BLACK or RED form after it has been created and the initial BLACK copy was returned to the application.

5 The newly generated RKEK is left in the specified key cache register, destkey, in the RED form.

10 The Generate RKEK command only generates Triple-DES secret keys. Once an RKEK is created, the secure Kernel only recognizes the RKEK as an internal key. Furthermore, the newly generated RKEK can only be used to cover/uncover application secret keys and public keys. Moreover, when an uncover operation is invoked to load an RKEK, the application is not allowed to specify any algorithm; the secure Kernel assumes triple DES in CBC mode. Also, the application is not allowed to provide a IV;, the secure Kernel provides it. This holds true for keys that are covered under an RKEK as well.

Command Interfaces:

```
20           /*generate an RKEK */
cgx_gen_rkek(kernelblock           *kb,
                  token_no_data   *t,
                  KCR             kcr
                  publickey       *dhpk
                  publickey       *dhkek
25           secretkey           *rkek)
```

Arguments:

```
30           (kb)->cb->cmd = CGX_GEN_RKEK; \
             (kb)->cb->argument[0].addr.dp = (UINT16)((kb)->dp); \
             (kb)->cb->argument[0].ptr = (VPTR)(t); \
             (kb)->cb->argument[1].ptr = (VPTR)(kcr); \
             (kb)->cb->argument[2].addr.dp = (UINT16)((kb)->dp); \
             (kb)->cb->argument[2].ptr = (VPTR)(dhpk); \
```

```

(kb)->cb->argument[3].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[3].ptr = (VPTR)(dhkek); \
(kb)->cb->argument[4].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[4].ptr = (VPTR)(rkek); \

```

5

Status:

```

kb->sb->status = CGX_SUCCESS_S,
               CGX_FAILED_RKEK_S,
               CGX_FAILED_GEN_RKEK_S.

```

10

See Also: CGX_UNCOVER_KEY and CGX_SAVE_KEY

SAVE KEY (Save a Key Under an RKEK)

15

Command Name: CGX_SAVE_KEY

Command Description:

The save key operation is a special operation used exclusively by an

RKEK. The save key operation uncovers a secret key under its context, then covers it under the RKEK.

20

Command Interfaces:

```

/*Save a key under an RKEK*/
cgx_save_key( kernelblock      *kb,
               secretkey       *bk_uncover,
               crypto_cntxt     *bkek,
               secretkey       *bk_returned,
               crypto_cntxt     *rkek)

```

25

30

Arguments:

```

(kb)->cb->cmd = CGX_SAVE_KEY; \
(kb)->cb->argument[0].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[0].ptr = (VPTR)(bk_uncovered); \
(kb)->cb->argument[1].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[1].ptr = (VPTR)(bkek); \
(kb)->cb->argument[2].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[2].ptr = (VPTR)(bk_returned); \

```

35

```
(kb)->cb->argument[3].addr.dp = (UINT16)((kb)->dp); \
(kb)->cb->argument[3].ptr = (VPTR)(rkek); \
```

5

Status:

```
kb->sb->status = CGX_SUCCESS_S,
                CGX_FAILED_SAVE_KEY_S.
```

10

See Also: CGX_UNCOVER_KEY and CGX_GEN_RKEK

GEN KEY (Generate A Secret Key)

Command Name: CGX_GEN_KEY

15

Command Description:

The Generate Key command allows the application to generate a user secret key. The secret key is generated using random numbers; then transformed into the secret key form as directed by the type of secret key specified (i.e. key_type) in the argument interface. The generated secret key can only be covered by an internal generated KEK (i.e. a GKEK) or via another user secret key (i.e. KKEK [if the generated key is an encryption key] or KEK), not the LSV.

20

Besides generating a secret key, this command places a copy of the key in its RED form in a KCR location and returns a BLACK copy of it all at one time; thus it is an 'intrinsic key' function. By returning a BLACK copy of the secret key immediately, another application is prevented from obtaining the secret key before the application responsible for its creation does so.

25

The application must also provide a key usage definition. This is passed in via the argument 'use' and is provided to announce the use of the newly generated secret key. Currently, the application has three

30

options: KKEK (CGX_KCR_KKEK) , KEK (CGX_KCR_KEK) or K (CGX_KCR_K). Defining it as KEK or KKEK directs the secure Kernel to only recognize this secret key as a KEK. Furthermore, when defined as a KKEK, the key cannot be used to cover anything other than data keys (type K). KEKs can only be used in key management operations (e.g. CGX_UNCOVER_KEY, etc.). Defining it as K directs the secure Kernel to only recognize it as a session key for traffic or message encryption operations (e.g. CGX_ENCRYPT, etc.).

A trust level must be given to a newly generated secret key. The user can force the secret key to be recognized as an untrusted secret key by or-ing in CGX_KCR_UNTRUSTED with the 'use' argument. This directs the secure Kernel to automatically mark the secret key as untrusted. This is useful if the application has plans to eventually export the secret key via the CGX_EXPORT_KEY command. Although, an untrusted secret key covered under a trusted KEK (i.e. GKEK, LSV, or a trusted KEK or KKEK) can not be exported; it is a securely protected key of the CryptIC device.

Moreover, if the application does not apply the un-trusted attribute to the 'use' argument, the secure Kernel will determine the trust level for the generated secret key. If the secret key is covered under a trusted KEK it will become a trusted key. If the secret key is covered under an untrusted KEK or KKEK it becomes an untrusted secret key. In other words, it inherits its parents trust attribute.

The newly generated secret key is left in the specified key cache register, destkey, in the RED form. A BLACK copy is returned if the application provides storage for a secretkey object and crypto_cntxt object. If either of these objects are NULL then no BLACK secret key

is returned. This is otherwise known as a dangling key. A dangling key is dynamic (one time or session oriented) because once the power is lost the secret key is lost as well. Furthermore, the key can not be exported because it has no covering KEK; this is required as authentication for exporting a secret key.

If storage for a BLACK secret key is provided then the newly generated secret key is covered by a KEK or KKEK specified in the crypto_cntxt object. The KCR location referenced in the crypto_cntxt object must point to a KCR that contains a KEK or KKEK or the operation fails. If the referenced KCR contains a KEK or KKEK the newly generated secret key is covered and returned to the application.

The Generate Key command only allows the generation of keys between 32bits and 168bits depending on the secret key type and if the state of the device (i.e. domestic or export). The application can choose between that range in increments of 8-bit units. Furthermore, the newly created user secret key can be used as an encryption/decryption key or as a user key encryption key (i.e. KEK) to cover other user secret keys.

Command Interfaces:

```
/*generate a secret key or KEK */
cgx_gen_key( kernelblock  *kb,
              kcr          destkey,
              unsigned short key_type,
              unsigned short length,
              unsigned short use,
              secretkey    *bk,
              crypto_cntxt *kek_cc)
```

Arguments:

```
kb->cb->cmd      = CGX_GEN_KEY;
```

```
/* KCR ID number to place newly generated user secret key */
kb->cb->argument[0] = (VPTR)destkey;
```

```

/* type of secret key to generate, use one of the following:
 * CGX_DES_A, and CGX_TRIPLE_DES_A.
 */
kb->cb->argument[1] = (VPTR)key_type;
5 /* length of user secret key to generate, 40 to 192 bits */
/* specified in units of bytes, in 8bit units */
kb->cb->argument[2] = (VPTR)length;
/* specify the key usage: CGX_KCR_K, CGX_KCR_KEK or CGX_KCR_KEK */
kb->cb->argument[3] = (VPTR)use;
10 /* storage for a BLACK secret key */
kb->cb->argument[4] = (VPTR)bk;
/* the KCR KEK location to cover the RED key into a BLACK key */
kb->cb->argument[5] = (VPTR)kek_cc;

15 Status:
kb->sb->status = CGX_SUCCESS_S,
                CGX_INVALID_REG_S,
                CGX_INVALID_KEY_LEN_S, or
                CGX_HARDWARE_FAILURE_S /* RNG hardware failure */
20

```

See Also: CGX_EXPORT_KEY, and CGX_UNCOVER_KEY

LOAD KEY (Import A RED User Secret Key)

25 **Command Name:** CGX_LOAD_KEY

Command Description:

30 The Load Key command is used to load a user secret key into a specified key cache register. The secret key to be imported is in the RED form, depending on the value of use, the key can be used as either a KEK or as an encryption key. This key is known as a user key to the secure Kernel and can never be covered by the LSV, the secure Kernel does not allow it.

35 After the secret key is loaded, if the user requested the black copy of the key (i.e., bk is non-NULL), the key is covered using the key encryption key specified in kek_cc. This is the only opportunity for the application to receive the black version of the key; the cover command no longer exists.

The secret key is to be loaded by using the secretkey data structure. The length field of the secretkey structure must be set by the application to indicate the length of the imported secret key. Salt is not required to be added to the key, it is in the RED form.

5

Command Interface:

```
/* import an application secret key */
cgx_load_key( kernelblock      *kb,
              kcr               key,
              secretkey         *sk,
              UINT16            use,
              secretkey         *bk,
              crypto_cntxt      *kek_cc)
```

10

Arguments:

15

```
kb->cb->cmd = CGX_LOAD_KEY;
```

20

```
/* key cache register ID to load key into */
kb->cb->argument[0] = (VPTR)key;
/* secretkey data structure pointer for key to be loaded */
kb->cb->argument[1] = (VPTR)sk;
/* kcr type to be used for loaded key (e.g. CGX_KEK_K) */
kb->cb->argument[2] = (VPTR)use;
/* pointer to buffer for storing BLACK key (NULL=>ignore) */
kb->cb->argument[3] = (VPTR)bk;
/* crypto context for the key encryption key */
kb->cb->argument[4] = (VPTR)kek_cc;
```

25

Status:

30

```
kb->sb->status = CGX_SUCCESS_S,
               CGX_FAIL_S, or
               CGX_INVALID_REG_S
```

See Also: CGX_UNCOVER_KEY

35

DERIVE KEY (Derive A Secret Key From a Pass Phrase)

Command Name: CGX_DERIVE_KEY

Command Description:

5 The Derive Key command allows a user secret key to be created from an application's pass-phrase. The secret key is derived by taking the one-way HASH of the application's pass phrase and using the message digest for it as the secret key bits; then transformed into the secret key form as directed by the type of secret key specified (i.e. key_type) in the argument interface. Furthermore, the derived secret key can only be covered by an internal generated KEK (i.e. a GKEK) or via another user secret key (i.e. KEK or KKEK, if the newly generated key is not a KEK), not the LSV.

10 The application can choose the HASH algorithm to be used via the argument, hash_alg. Furthermore, the algorithm for choosing which bits to use is outlined in Microsoft's CryptoAPI document, this command is implemented to their specification. Also, the algorithm only supports key bit lengths between 32 bits and 112 bits when
15 creating DES or Triple DES keys, 32 bits through 160 bits when creating HMAC keys, and 32 bits through 128 bits when creating RC5 keys.

20 Besides deriving a secret key this command places a copy of the RED form in a KCR location and returns a BLACK copy of it all at one time; thus an intrinsic key function. By returning a BLACK copy of the secret key immediately another application is prevented from grabbing the secret key before the application responsible for its creation does so.

25 The application must also provide a key usage definition. This is passed in via the argument, use, and is provided to announce the use of the newly derived secret key. Currently, the application has three options: KEK (CGX_KCR_KEK), KKEK (CGX_KCR_KKEK) or K

(CGX_KCR_K). Defining it as KEK directs the secure Kernel to only recognize this secret key as a KEK. Defining the key as a KKEK restricts the use of the key to only covering other keys of type K (CGX_KCR_K). Therefore, these keys can only be used in key management operations (e.g. CGX_UNCOVER_KEY, etc.). Defining it as K directs the secure Kernel to only recognize it as a session key for traffic or message encryption operations (e.g. CGX_ENCRYPT, etc.).

Furthermore, a trust level must be given to a newly derived secret key. The user can force the secret key to be recognized as an untrusted secret key by or-ing in CGX_KCR_UNTRUSTED with the use argument. This directs the secure Kernel to automatically mark the secret key as untrusted. This is useful if the application has plans to eventually export the secret key via the CGX_EXPORT_KEY command. Although, an untrusted secret key covered under a trusted KEK (i.e. GKEK, LSV, or a trusted KEK or KKEK) can not be exported; it's a securely protected key of the CryptIC device.

Moreover, if the application does not apply the un-trust attribute to the use argument the secure Kernel will determine the trust level for the derived secret key. Unlike the secret keys generated via the CGX_GEN_KEY command all secret keys derived with this command are set as untrusted by the secure Kernel.

The newly derived secret key is left in the specified key cache register, destkey, in the RED form. A BLACK copy is returned if the application provides storage for a secretkey object and crypto_cntxt object. If either of these objects are NULL then no BLACK secret key is returned. This is otherwise known as a dangling key. A dangling key is dynamic (one time or session oriented) because once the power is

lost the secret key is as well. Furthermore, the key can not be exported because it has no covering KEK, this is required as authentication for exporting a secret key.

If storage for a BLACK secret key is provided then the newly derived secret key is covered by a KEK specified in the crypto_cntxt object. The KCR location referenced in the crypto_cntxt object must point to a KCR that contains a KEK or the operation fails. If the referenced KCR contains a KEK the newly derived secret key is covered and returned to the application.

The Derive Key command only allows the generation of keys between 32-bits and 112-bits depending on the secret key type and if the state of the device (i.e. domestic or export). The application can choose between that range in increments of 8bit units. Furthermore, the newly created user secret key can be used as an encryption/decryption key or as a user key encryption key (i.e. KEK) to cover other user secret keys.

Command Interfaces:

```
/*derive a secret key */
cgx_gen_key( kernelblock  *kb,
             unsigned short pswd_pg,
             unsigned short *pswd,
             unsigned short pswd_len,
             unsigned short hash_alg,
             kcr          destkey,
             unsigned short key_type,
             unsigned short length,
             unsigned short use,
             secretkey     *bk,
             crypto_cntxt  *kek_cc)
```

Arguments:

```
kb->cb->cmd      = CGX_DERIVE_KEY;
```

```

/* the application's pass phrase string */
kb->cb->argument[0] = (VPTR)pswd;
/* the length of the application's pass phrase string in bytes */
kb->cb->argument[1] = (VPTR)pswd_len;
5  /* the HASH alg to use: CGX_SHS_A, or CGX_MD5_A */
kb->cb->argument[2] = (VPTR)hash_alg;
/* KCR ID number to place newly generated user secret key */
kb->cb->argument[3] = (VPTR)destkey;
/* type of secret key to generate, use one of the following:
10  * CGX_DES_A, and CGX_TRIPLE_DES_A.
*/
kb->cb->argument[4] = (VPTR)key_type;
/* length of user secret key to generate, 40 to 192 bits */
/* specified in units of bytes, in 8bit units */
15 kb->cb->argument[5] = (VPTR)length;
/* specify the key usage: CGX_KCR_K, CGX_KCR_KKEK, or CGX_KCR_KEK */
kb->cb->argument[6] = (VPTR)use;
/* storage for a BLACK secret key */
kb->cb->argument[7] = (VPTR)bk;
20 /* the KCR KEK location to cover the RED key into a BLACK key */
kb->cb->argument[8] = (VPTR)kek_cc;
/* For the ADI target only, the data page where the pswd resides */
kb->cb->argument[9] = (VPTR)pswd_pg;

25  Status:
kb->sb->status = CGX_SUCCESS_S,
                CGX_INVALID_REG_S,
                CGX_INVALID_KEY_LEN_S, or
                CGX_HARDWARE_FAILURE_S /* RNG hardware failure */
30

```

See Also: CGX_EXPORT_KEY, and CGX_UNCOVER_KEY

TRANSFORM KEY (Transform A Secret Key Using IPsec)

Command Name: CGX_TRANSFORM_KEY

35 **Command Description:**

The Transform Key command allows the application to perform one of the IPsec secret key transforms on an existing secret key. The transform command allows the application to create the HMAC, CBC DES, or CBC Triple DES keys. Furthermore, it can be used to create the IV and replay counters and beyond that it can be used to create the HMAC inner and outer pre-computed digests to speed up the AH processing.

40

There are several variants of this command. This section describes the common function that supports them all. The **Command Interface** subsections describe macros that invoke the common function to achieve a specific variation of the command. Users are urged to employ the specific macros to invoke the command variants.

Any untrusted or trusted secret key, K, can be transformed. At no time can a trusted or untrusted KEK, KKEK, GKEK, or LSV be used as the root secret key to be transformed. The result can be returned as a new covered BLACK secret key or as a clear pre-computation of a secret key. However, in either case, the root secret key remains intact - it is read-only.

Besides transforming a secret key, this command places a copy of the key in its RED form (if an HMAC or CBC DES key is to be generated) in a KCR location and returns a BLACK copy of it all at one time; thus it is an intrinsic key function. By returning a BLACK copy of the secret key immediately, another application is prevented from grabbing the secret key before the application responsible for its creation does so.

In the case of generating another secret key the key usage definition of the root secret key is inherited. Therefore, the newly generated secret key will inherit the trust level (i.e. trusted or untrusted) and key usage (i.e. K). However, the application can change the key type to any valid supported key (i.e. DES, triple DES, or RC5) via the argument, ktype. Furthermore, it can generate any key length it desires via the argument, klen.

The newly transformed secret key is left in the specified key cache register (kcr), destkey, in the RED form. A BLACK copy is returned if the application provides storage for a secretkey object, tk. It is covered under the

root secret key's KEK which is passed in as the crypto_cntxt object argument, bkek. If the secretkey object argument, tk, is passed in as a NULL pointer, the operation will return a message digest via the hash_context object argument, hc. This returned hash context may be red or black (covered.) If the caller
5 wishes the returned hash context to be black, the user must specify a crypto context, hkek, which the command will use to cover the returned hash context, hc. If the user supplies a NULL parameter for hkek, hc will be returned in the red (uncovered.)

In order to support the current IPsec transform, the application must
10 pass-in the patterns and operation data as arguments. The 'patterns' are the data patterns to be applied (ie. XORed) to a root secret key and then HASHed. Although the IPsec patterns are currently fixed to a single byte that is repeated 64 times, this command forces the application to pass in the full 64-byte patterns required to create a transformed secret key. This allows for the
15 possibility of future changes.

The application must pass in as many 64-byte patterns (i.e. as one large array) as they will generate key bits or as long as specified by the argument, klen. If the key to transform is to be a DES key and klen is less then or equal to 7 bytes, then 64 bytes of pattern are required, if the klen is less than or equal
20 to 14 bytes then 128 bytes - or two 64 byte patterns - in one array are required, and if klen is less than or equal to 21 bytes then 192 bytes - or three 64 byte patterns - in one array are required. If the key to transform is an RC5 key, then the application is only required to pass-in one 64 byte pattern.

The operation argument, oper, is used to specify the operation which
25 will 'combine' the root secret key and the data patterns. Currently, the command supports three operations: exclusive-or (i.e. CGX_XOR_O), pattern

append (i.e. CGX_APPEND_O), and pattern pre-pend (i.e. CGX_PREPEND_O).

Command Interfaces:

/*transform a root secret key into a DES or HMAC key */

/* Common interface. Caller must select proper combination of parameters to achieve desired functionality. */

/* This interface is not recommended for user applications; instead user should use one of the subsequently described variants */

```
cgx_transform_key_common( kernelblock *kb,
                          secretkey    *bk,
                          crypto_cntxt *bkek,
                          secretkey    *tk,
                          unsigned short ktype,
                          unsigned short klen,
                          unsigned short *patterns,
                          unsigned short oper,
                          unsigned short halg,
                          hash_cntxt    *hc,
                          crypto_cntxt  *hkek);
```

Arguments:

kb->cb->cmd = CGX_TRANSFORM_KEY;

/* the BLACK root secret key */

kb->cb->argument[0] = (VPTR)bk;

/* the KCR KEK location to uncover the root BLACK key */

kb->cb->argument[1] = (VPTR)bkek;

/* storage for a new BLACK transformed secret key */

kb->cb->argument[2] = (VPTR)tk; /* see text above */

/* KCR ID number to place newly generated transformed secret key */

kb->cb->argument[3] = (VPTR)destkey;

/* type of secret key to generate, use one of the following:

* CGX_RC5_A, CGX_DES_A, and CGX_TRIPLE_DES_A
*/

kb->cb->argument[4] = (VPTR)ktype;

/* length of user secret key to generate, 40 to 168 bits */

/* specified in units of bytes, in 8-bit units (5 - 21) */

kb->cb->argument[5] = (VPTR)klen;

/* transform 64 byte data patterns */

kb->cb->argument[6] = (VPTR)patterns;

/* the data pattern transform operation, XOR, APPEND, PREPEND */

kb->cb->argument[7] = (VPTR)oper;

/* the HASH alg to use: CGX_SHS_A, or CGX_MD5_A */

kb->cb->argument[8] = (VPTR)hash_alg;

/* the hash_cntxt to return the pre-comp of HMAC key */

kb->cb->argument[9] = (VPTR)hc;

kb->cb->argument[10] = (VPTR)hkek; /* see text above */

Status:

```

kb->sb->status = CGX_SUCCESS_S,
                  CGX_FAIL_S, and
                  CGX_INVALID_REG_S

```

```

/* This variant is preserved only for backward compatibility. One of the next three variants to be
described is recommended for new development. This variant supplies a NULL hkek parameter to the
common function, thus requesting that any returned hash context (in hc) will be red */

```

```

cgx_transform_key( kernelblock    *kb,
                   secretkey      *bk,
                   crypto_cntxt   *bkek,
                   secretkey      *tk,
                   unsigned short  ktype,
                   unsigned short  klen,
                   unsigned short  *patterns,
                   unsigned short  oper,
                   unsigned short  halg,
                   hash_cntxt      *hc)

```

This variant is defined as:

```

cgx_transform_key_common( (kb), (bk), (bkek), (tk), (ktype), (klen),
                          (patterns), (oper), (hash_alg), (hc), NULL)

```

```

/* variant: Transform a root secret key into a pre-computed HMAC hash context returned in the red */

```

```

cgx_transform_precompute_key( kernelblock *kb,
                              secretkey   *bk,
                              crypto_cntxt *bkek,
                              unsigned short final, /* close has context if final == TRUE */
                              unsigned short *patterns,
                              unsigned short oper,
                              unsigned short hash_alg,
                              hash_cntxt   *hc)

```

This variant is defined as:

```

cgx_transform_key_common( (kb), (bk), (bkek), NULL, NULL,
                          (final), (patterns), (oper), (hash_alg), (hc), NULL)

```

/* variant: Transform a root secret key into a black key. */

```

5  cgx_transform_gen_key( kernelblock  *kb,
                        secretkey    *bk,
                        crypto_cntxt *bkek,
                        secretkey    *tk,
                        unsigned short ktype,
10  unsigned short klen,
                        unsigned short *patterns,
                        unsigned short oper,
                        unsigned short hash_alg)

```

This variant is defined as:

```

cgx_transform_key_common( (kb), (bk), (bkek), (tk), (ktype), (klen),
(patterns), (oper), (hash_alg), NULL, NULL)

```

/* variant: Transform a root secret key into a pre-computed HMAC hash context returned in the black*/

```

20  cgx_transform_precomputed_bkey( kernelblock  *kb,
                        secretkey    *bk,
                        crypto_cntxt *bkek,
                        unsigned short final,
25  unsigned short *patterns,
                        unsigned short oper,
                        unsigned short hash_alg,
                        hash_cntxt    *hc,
                        crypto_cntxt  *hkek);
30

```

This variant is defined as:

```

cgx_transform_key_common( (kb), (bk), (bkek), NULL, NULL,
35  (final), (patterns), (oper), (hash_alg), hc, hkek)

```

See Also: CGX_HASH_ENCRYPT or CGX_HASH_DECRYPT

EXPORT KEY (Export An IRE Secret Key)**Command Name:** CGX_EXPORT_KEY**Command Description:**

5 The Export Key command allows the application to move an IRE secret key form into an external secret key form. The external secret form must be covered either with a secret key or public key, this is specified by the application via the command arguments.

10 The application must present a BLACK copy of the IRE secret key to export along with the crypto_cntxt object to reference the KEK or KKEK to uncover it. Also, the application must provide a buffer (i.e. ebk) to copy the external secret key into that is converted from the BLACK IRE secret key along with a crypto_cntxt object referencing a KEK to cover it or a publickey object to cover it with.

15 The IRE secret key to export must be an untrusted key, KKEK or KEK. Furthermore, it must not reside (or covered by) under a parent KEK that is trusted (i.e. LSV, GKEK, or trusted KEK or KKEK). Moreover, the secure attributes stored in each of the IRE BLACK secret keys are removed before the secret key is exported.

20 These bits are not used or would not be understood in other vendor's crypto equipment. Therefore, the main purpose of this command is to provide some sort of key interoperability between an IRE crypto device and some other vendor's crypto equipment (software or hardware based).

25 As part of the command the application is allowed to program salt bits that will be prepended to the secret key bits of the external secret key. When exporting a secret key under another secret key the

application has the choice of providing salt bits in multiples of 2 bytes, requesting the secure Kernel generate salt bits in multiples of 2 bytes, or not storing any salt bits. However, the total of salt bytes, data bytes, and the key bytes must be a multiple of 8 bytes. When covering under a public key it can request salt in multiples of 2 bytes and the total of salt, data, and secret key only has to be a multiple of 2 bytes and less than the modulus length; at least make sure its most significant bit is not set.

Also, the command allows the application to program data bits that will be appended to the secret key bits of the external secret key. When exporting a secret key under another secret key the application has the choice of providing data bits in multiples of 2 bytes, requesting the secure Kernel generate data bits in multiples of 2 bytes, or not storing any data bits. However, the total of salt bytes, data bytes, and the key bytes must be a multiple of 8 bytes. When covering under a public key it can request data in multiples of 2 bytes and the total of salt, data, and secret key only has to be a multiple of 2 bytes.

The command does not store all keys in multiples of 8 bytes. For example the HMAC and RC5 keys may not be exported out in an 8 byte multiple if its size is not of 8 byte multiples. However, DES and Triple DES keys will always be exported in multiples of 8 bytes. Furthermore the secret key is flipped when exported, it is put into big endian order.

All DES and Triple DES keys follow these storage rules. If the key is less than or equal to 7 bytes in length then it is exported as 8 bytes, if the key is less than or equal to 14 bytes in length then it is exported as 16 bytes, and if the key is less than or equal to 21 bytes in length it is exported as 24 bytes. The expansion in key size is due to the DES parity bits.

All RC5 keys follow these storage rules. If the key is odd size in length an extra byte is added to make it a multiple of 2; the extra byte is a 0. Otherwise, the RC5 key is exported out as is, its programmed length. The zero is appended to key so that the 0 can be used by Microsoft CAPI CSPs as a key delimiter.

All HMAC keys follow these storage rules. If the key is odd size in length an extra byte is added to make it a multiple of 2; the extra byte is a 0. Otherwise, the HMAC key is exported out as is, its programmed length. The zero is appended to key so that the 0 can be used by Microsoft CAPI CSPs as a key delimiter.

If the argument, salt_len, is set to 0 the application requests that no salt bits be prepended. If the argument, salt, is NULL it requests that the secure Kernel generate salt_len bytes; otherwise the application is providing salt_len bytes of salt.

If the argument, data_len, is set to 0 the application requests that no data bits be prepended. If the argument, data, is NULL it requests that the secure Kernel generate data_len bytes; otherwise the application is providing data_len bytes of data.

If the publickey argument, pk, is NULL then the secret key to export is covered with the KEK, ekek_cc. If pk is not NULL then the secret key to export is covered with the public key pk object.

Command Interfaces:

```
/*export an IRE secret key under another untrusted secret key */
cgx_export_key( kernelblock      *kb,
                secretkey       *bk,
                crypto_cntxt     *bkek_cc,
                unsigned short   *ebk,
                crypto_cntxt     *ekek_cc,
                unsigned short   *salt,
                unsigned short salt_len,
```

```

        unsigned short    *data,
        unsigned short data_len)

```

Arguments:

```

5      kb->cb->cmd          = CGX_EXPORT_KEY;

        /* the BLACK IRE secret key to export */
        kb->cb->argument[0] = (VPTR)bk;
        /* the KCR KEK location to uncover the BLACK IRE secret key */
10     kb->cb->argument[1] = (VPTR)bkek_cc;
        /* the buffer to house the external secret key */
        kb->cb->argument[2] = (VPTR)ebk;
        /* the KCR KEK location to cover the RED exported IRE secret key */
        kb->cb->argument[3] = (VPTR)ekek_cc;
15     /*salt bits to prepend to key bits */
        kb->cb->argument[4] = (VPTR)salt;
        /* the number of salt bits to prepend to the key bits */
        /* this must be in units of 2bytes */
        kb->cb->argument[5] = (VPTR)salt_len;
20     /*data bits to prepend to key bits */
        kb->cb->argument[6] = (VPTR)data;
        /* the number of data bits to prepend to the key bits */
        /* this must be in units of 2bytes */
        kb->cb->argument[7] = (VPTR)data_len;
25     /* pubkey key to cover with */
        kb->cb->argument[8] = (VPTR)NULL;

        /*export an IRE secret key under a public key */
        cgx_export_key( kernelblock *kb,
30                      secretkey    *bk,
                      crypto_cntxt  *bkek_cc,
                      unsigned short *ebk,
                      publickey     *pk,
                      unsigned short *salt,
35                      unsigned short salt_len,
                      unsigned short *data,
                      unsigned short data_len)

```

Arguments:

```

40     kb->cb->cmd          = CGX_EXPORT_KEY;

        /* the BLACK IRE secret key to export */
        kb->cb->argument[0] = (VPTR)bk;
        /* the KCR KEK location to uncover the BLACK IRE secret key */
45     kb->cb->argument[1] = (VPTR)bkek_cc;
        /* the buffer to house the external secret key */
        kb->cb->argument[2] = (VPTR)ebk;
        /* the KCR KEK location to cover the RED exported IRE secret key */
        kb->cb->argument[3] = (VPTR)NULL;

```

```

    /*salt bits to prepend to key bits */
    kb->cb->argument[4] = (VPTR)salt;
    /* the number of salt bits to prepend to the key bits */
    /* this must be in units of 2bytes */
5    kb->cb->argument[5] = (VPTR)salt_len;
    /*data bits to prepend to key bits */
    kb->cb->argument[6] = (VPTR)data;
    /* the number of data bits to prepend to the key bits */
    /* this must be in units of 2bytes */
10   kb->cb->argument[7] = (VPTR)data_len;
    /* pubkey key to cover with */
    kb->cb->argument[8] = (VPTR)pk;

```

Status:

```

15   kb->sb->status = CGX_SUCCESS_S, or
                        CGX_INVALID_REG_S

```

20 **See Also:** CGX_GEN_KEY, CGX_DERIVE_KEY, CGX_LOAD_KEY, and CGX_IMPORT_KEY

IMPORT KEY (Import An IRE Secret Key)

Command Name: CGX_IMPORT_KEY

25 **Command Description:**

The Import Key command allows the application to load an IRE secret key form in external secret key form into the secure kernel.

30 This command allows the application to import a key into the specified key cache register. The application must present the external form of the key to be imported, ibk, and specify the length of the salt prepended to the key through the salt_len parameter. The salt length is assumed to be a multiple of 2 bytes, and can be 0 bytes. The crypto_cntxt, ikek_cc, will be used to decrypt the salt (if present), key and any additional padding.

35 The application controls the algorithm type, key_type, and length of the key being imported. The application also specifies how

the key is to be used, e.g. encryption key or key encryption key. However, the kernel makes no assumptions about the security of the key being imported. Therefore, upon completion of the command, the newly imported key is treated as an untrusted key.

5 The bk parameter can be used to request a covered version of the key once it has been successfully imported. If bk is non-NULL, and the import succeeds, the key will be covered under the crypto_cntxt pointed to by kek_cc and stored at the location specified by bk.

Command Interfaces:

```
10           /*import an IRE secret key */
          cgx_import_key( kernelblock *kb,
                          secretkey     *ibk,
                          UINT16       salt_len,
                          crypto_cntxt *ikek_cc,
15           kcr         dest_kcr,
                          UINT16       key_type,
                          UINT16       length,
                          UINT16       use,
                          secretkey     *bk,
20           crypto_cntxt *kek_cc)
```

Arguments:

```
          kb->cb->cmd         = CGX_IMPORT_KEY;
25           /* the secret key blob to import */
          kb->cb->argument[0] = (VPTR)ibk;
          /* the length of the salt before the key material */
          kb->cb->argument[1] = (VPTR)salt_len;
30           /* the KCR KEK location used to uncover the imported secret key */
          kb->cb->argument[2] = (VPTR)ikek_cc;
          /* the destination KCR location for storing the imported key */
          kb->cb->argument[3] = (VPTR)dest_kcr;
          /* the algorithm type of key being imported */
35           kb->cb->argument[4] = (VPTR)key_type;
          /* the KCR KEK location to cover the imported IRE secret key */
          kb->cb->argument[5] = (VPTR)length;
          /* how the imported key will be used in the key hierarchy
          *(e.g. CGX_KCR_K)
40           */
          kb->cb->argument[6] = (VPTR)use;
          /* location to store resulting covered secretkey (NULL=>ignore) */
```



```

kb->cb->argument[7] = (VPTR)bk;
/* the KCR KEK location to cover the imported IRE secret key */
kb->cb->argument[8] = (VPTR)kek_cc;

```

5

Status:

```

kb->sb->status = CGX_SUCCESS_S,
                CGX_INVALID_REG_S,
                CGX_EMPTY_REG_S,
                CGX_INVALID_LEN_S, or
                CGX_FAIL_S

```

10

See Also: CGX_GEN_KEY, CGX_DERIVE_KEY, CGX_LOAD_KEY, and CGX_EXPORT_KEY

15

DESTROY KEY (Remove Secret Key From KCR)

Command Name: CGX_DESTROY_KEY

Command Description:

20

The Destroy Key command is used to remove a secret key from one the specified key cache register.

25

Access to the secret keys is via key cache register IDs. The key cache register IDs are numbered from 0 to N, there are N secret key registers available to the application. Key cache register 0 is reserved for the LSV and, therefore, cannot be destroyed. Therefore, the key cache register available to the application range are from 1 to N.

If the KCR is already empty, the command will report back as though it successfully removed the secret key from the KCR.

Command Interface:

30

```

/* destroy a secret key in either the FLASH or volatile KCR areas */
cgx_destroy_key( kernelblock *kb, kcr key )

```

Arguments:

35

```

kb->cb->cmd = CGX_DESTROY_KEY;

```

```
/* the KCR ID to remove */  
kb->cb->argument[0] = (VPTR) key;
```

5

Status:

```
kb->sb->status = CGX_SUCCESS_S, or  
CGX_INVALID_REG_S /* KCR invalid id */
```

10

See Also: CGX_LOAD_KEY**LOAD KG (Load Secret Key Into HW/SW KG)****Command Name:** CGX_LOAD_KG

15

Command Description:

20

The Load KG command is used to load DES/Triple DES secret keys into the hardware key generator and RC5 keys into the RC5 software key generator. The typical use of this command is to fully optimize secret key traffic. For example, loading the KG once and using the static kernel block approach will speed the encryption process because of fewer secret key context switches.

25

Prior to invoking the Load KG command, it is preferred if the application sets up the crypto_cntxt block. The application is responsible for setting the mode, setting the appropriate secret key KCR ID, and for priming the iv buffer with an initial random number (if this is the first time the crypto_cntxt is loaded). After this the application need not modify the crypto_cntxt block unless something changes (i.e., secret key KCR ID location, and/or the iv to cause a resynchronization).

30

The iv buffer of the crypto_cntxt block must be read- and write-able; the iv buffer is used to maintain the feed-back register for the CBC, CFB, and OFB modes. In the ECB mode the iv buffer is ignored.

5 The secret key to be loaded as referenced via the `crypto_cntxt` must have the key usage setting of `CGX_KCR_K`. This command only allows traffic or data keys to be loaded or it will fail. The key usage information is programmed by the application at the time a secret key has been loaded, generated, derived, or negotiated and is maintained securely by the secure Kernel. This means an LSV, GKEK, or KEK type of secret key can not be loaded by this command.

10 If a KKEK (i.e. DES or Triple DES only) is loaded it is loaded into the special hardware KKEK register to be used to externally uncover application BLACK secret keys.

Command Interface:

```

15      /* load a single secret key into the SW/HW KG */
      cgx_load_kg( kernelblock  *kb,
                   crypto_cntxt *cb,
                   unsigned short direction)

      /* load multiple secret keys for triple DES into the HW KG */

20      Arguments:
          kb->cb->cmd          = CGX_LOAD_KG;

          /* pointer to the crypto_cntxt to load into the HW/SW KG */
          kb->cb->argument[0] = (VPTR)cb;
25      // direction is used to specify the use of the KG,
          // encrypt or decrypt. To specify the encrypt operation direction
          // must be set to non-0, for decrypt then direction must be set
          // to 0
          kb->cb->argument[1] = (VPTR)direction;
30

      Status:
          kb->sb->status = CGX_SUCCESS_S,
                        CGX_WEAK_KEY_S,
                        CGX_EMPTY_REG_S, or
35                        CGX_INVALID_REG_S

```

See Also: CGX_ENCRYPT, CGX_DECRYPT, CGX_HASH_ENCRYPT, CGX_HASH_DECRYPT, or CGX_STREAM

ENCRYPT (Encrypt Data)

5 **Command Name:** CGX_ENCRYPT

Command Description:

10 The Encrypt command is used to perform symmetrical encryption. The encrypt operation supports many secret key algorithms (i.e., DES, Triple DES, and RC5) in any of the modes (i.e., ECB, CFB, OFB, and CBC).

15 The Encrypt command only supports block encryption, a block must be 64 bits long. The Encrypt command can handle as many blocks as the application chooses to encrypt at one time. Furthermore, an encrypted data session can extend beyond one call to the encryption command; this is accomplished via a crypto_cntxt block (described below).

20 The data buffers, datain (plain-text) and dataout (cipher-text), can share the same address. Moreover, the chip currently only supports cipher-block symmetrical algorithms but the encrypt interface provides for the addition of a stream interface. It does this by defining datain and dataout as unsigned char pointers and the data length of datain to be in bytes. Allowing the byte count makes the interface portable for the later addition of stream based algorithms. However, if the algorithm to be used in the encrypt operation is cipher-block based the
25 byte count must be evenly divisible by 8, any fragments are ignored by the operation.

Prior to invoking the encryption command the application must setup the `crypto_cntxt` block. The application is responsible for setting the mode, masking in one of the special key load options (described below), setting the appropriate secret key KCR ID, and for priming the iv buffer with an initial random number (only for the first call). After this the application need not modify the `crypto_cntxt` block unless something changes (i.e., secret key KCR ID location, algorithm, and/or the iv to cause a resynchronization).

The KCR key specified in the `crypto_cntxt` contains the algorithm to use as part of the secret key object stored in the specified KCR location. This secret key type specifies the secret key algorithm to use.

The iv buffer of the `crypto_cntxt` block must be read and writeable; the iv buffer is used to maintain the feed-back register for the CBC, CFB, and OFB modes. In the ECB mode the iv buffer is ignored.

The Encrypt command allows the application to mask in one of the special algorithm and mode control bits defined in section 0, into the algorithm field of the `crypto_cntxt` block. The control bits are used by the secure Kernel to determine how to load secret keys before the actual encryption of the plain-text takes place. The control bits allow the application to request one of these options: auto-load, force-load, or no-load. Auto-load allows the secure Kernel to check which key is currently loaded into the KG, if it's the same as the key specified for the encryption command it is not loaded; otherwise the key is loaded. Force-load tells the secure Kernel to always load the key. No-load tells the secure Kernel to not load the key; more than likely the application

has already loaded the key (maybe via the CGX_LOAD_KG command). By default, auto-load is assumed.

The secret key to be loaded as referenced via the crypto_cntxt must have the key usage setting of CGX_KCR_K. This command only allows traffic or data keys to be loaded or it will fail. The key usage information is programmed by the application at the time a secret key has been loaded, generated, derived, or negotiated and is maintained securely by the secure Kernel. This means an LSV, GKEK, or KEK type of secret key can not be loaded by this command.

When using the CFB or OFB modes the application can signal the start of secure traffic or resynchronize it by setting the CGX_RESYNC_C control bit. This will signal the CGX_ENCRYPT command to first encrypt the application's IV before encrypting the input data. This command does not clear the bit so upon return the application should clear the CGX_RESYNC_C bit before invoking the command again or the IV resynchronization will occur again.

Command Interface:

```
cgx_encrypt( kernelblock  *kb,
              unsigned short datain_page,
              unsigned char *datain,
              unsigned short dataout_page,
              unsigned char *dataout,
              unsigned      byte_cnt,
              crypto_cntxt *cb)
```

Arguments:

```
kb->cb->cmd      = CGX_ENCRYPT;
```

```
/* mode selection,
 * mask one of the special algorithm
 * control bits with the mode here via the crypto_cntxt block
 */
```

```
kb->cb->argument[0] = (VPTR)cb;
/* data block(s) to be encrypted data page, only needed */
```

```
/* for the ADI 2181 platform, all other platform ignore it */
kb->cb->argument[1] = (VPTR)datain_page;
/* data block(s) to be encrypted */

5 kb->cb->argument[2] = (VPTR)datain;
/* specified in units of bytes, in the current case all the */
/* algorithms use 64 bit blocks so the value of byte_cnt must */
/* be evenly divisible by 8, any fragments are ignored */
10 kb->cb->argument[3] = (VPTR)byte_cnt;
/* data block(s) to be encrypted data page, only needed */
/* for the ADI 2181 platform, all other platform ignore it */
kb->cb->argument[4] = (VPTR)dataout_page;
/* output the encrypted data blocks */
15 kb->cb->argument[5] = (VPTR)dataout;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S,
                CGX_WEAK_KEY_S,
20 CGX_EMPTY_REG_S, or
                CGX_INVALID_REG_S
```

See Also: CGX_LOAD_KG and CGX_DECRYPT

25

DECRYPT (Decrypt Data)

Command Name: CGX_DECRYPT

Command Description:

5 The Decrypt command is used to perform symmetrical decryption. The decrypt operation supports many secret key algorithms (i.e., DES, Triple DES, and RC5) in any of the modes (i.e., ECB, CFB, OFB, and CBC).

10 The Decrypt command only supports block decryption, a block must be 64 bits long. The Decrypt command can handle as many blocks as the application chooses to decrypt at one time. Furthermore, a decrypted data session can extend beyond one call to the decryption command; this is accomplished via a crypto_cntxt block (described below).

15 The data buffers, datain (plain-text) and dataout (cipher-text), can share the same address. Moreover, the chip currently only supports cipher-block symmetrical algorithms but the decrypt interface provides for the addition of a stream interface. It does this by defining datain and dataout as unsigned char pointers and the data length of datain to be in bytes. Allowing the byte count makes the interface portable for
20 the later addition of stream based algorithms. However, if the algorithm to be used in the decrypt operation is cipher-block based the byte count must be evenly divisible by 8, any fragments are ignored by the operation.

25 Prior to invoking the Decrypt command the application must setup the crypto_cntxt block. The application is responsible for setting the mode, masking in one of the special key load options (described below), setting the appropriate secret key KCR ID, and for priming the

iv buffer with an initial random number (only for the first call). After this the application need not modify the `crypto_cntxt` block unless something changes (i.e., secret key KCR ID location, algorithm, and/or the iv to cause a resynchronization).

5 The KCR key specified in the `crypto_cntxt` contains the algorithm to use as part of the secret key object stored in the specified KCR location. This secret key type specifies the secret key algorithm to use.

10 The iv buffer of the `crypto_cntxt` block must be read and writeable; the iv buffer is used to maintain the feed-back register for the CBC, CFB, and OFB modes. In the ECB mode the iv buffer is ignored.

15 The Decrypt command allows the application to mask in one of the special algorithm and mode control bits defined in section 0, into the algorithm field of the `crypto_cntxt` block. The control bits are used by the secure Kernel to determine how to load secret keys before the actual encryption of the plain-text takes place. The control bits allow the application to request one of these options: auto-load, force-load, or no-load. Auto-load allows the secure Kernel to check which key is currently loaded into the KG, if its the same as the key specified for the encryption command it is not loaded; otherwise the key is loaded.

20 Force-load tells the secure Kernel to always load the key. No-load tells the secure Kernel to not load the key; more than likely the application has already loaded the key (maybe via the `CGX_LOAD_KG` command). By default, auto-load is assumed.

25 The secret key to be loaded as referenced via the `crypto_cntxt` must have the key usage setting of `CGX_KCR_K`. This command only allows traffic or data keys to be loaded or it will fail. The key usage information is programmed by the application at the time a secret key

has been loaded, generated, derived, or negotiated and is maintained securely by the secure Kernel. This means an LSV, GKEK, or KEK type of secret key can not be loaded by this command.

When using the CFB or OFB modes the application can signal the start of secure traffic or resynchronize it by setting the CGX_RESYNC_C control bit. This will signal the CGX_DECRYPT command to first encrypt the application's IV before decrypting the input data. This command does not clear the bit so upon return the application should clear the CGX_RESYNC_C bit before invoking the command again or the IV resynchronization will occur again.

Command Interface:

```
cgx_decrypt( kernelblock  *kb,
              unsigned short datain_page,
              unsigned short *datain,
              unsigned short dataout_page,
              unsigned short *dataout,
              unsigned   byte_cnt,
              crypto_cntxt *cb)
```

Arguments:

```
kb->cb->cmd      = CGX_DECRYPT;
```

```
/* mode selection,
```

```
   * mask one of the special algorithm
```

```
   * control bits with the mode here via the crypto_cntxt block
```

```
*/
```

```
kb->cb->argument[0] = (VPTR)cb;
```

```
/* data block(s) to be decrypted data page, only needed */
```

```
/* for the ADI 2181 platform, all other platform ignore it */
```

```
kb->cb->argument[1] = (VPTR)datain_page;
```

```
/* data block(s) to be decrypted */
```

```
kb->cb->argument[2] = (VPTR)datain;
```

```
/* specified in units of blocks, in the current case all the */
```

```
/* algorithms use 64 bit blocks so the value of byte_cnt must */
```

```
/* be evenly divisible by 8, any fragments are ignored */
```

```
kb->cb->argument[3] = (VPTR)byte_len;
```

```
/* data block(s) to be decrypted data page, only needed */
```

```
/* for the ADI 2181 platform, all other platform ignore it */
```

```
kb->cb->argument[4] = (VPTR)dataout_page;
```

```
/* output the decrypted data blocks */
kb->cb->argument[5] = (VPTR)dataout;
```

Status:

```
5 kb->sb->status = CGX_SUCCESS_S,
                  CGX_WEAK_KEY_S,
                  CGX_EMPTY_REG_S, or
                  CGX_INVALID_REG_S
```

10 **See Also:** CGX_LOAD_KG and CGX_ENCRYPT

15

Public Key Commands

GEN PUBKEY (Generate a Public Keyset)

Command Name: CGX_GEN_PUBKEY

Command Description:

20 This operation will generate an entire public keyset comprised of the modulus, private, and public blocks. This operation can create public keysets for several public key algorithms. It currently supports: Diffie-Hellman, RSA, and DSA public keys. The returned keyset will consist of data stored in little endian order.

25 The newly generated public keyset is returned to the application via the publickey parameter. The modulus and public key are returned in the RED form while the private key is returned in the BLACK form covered by the specified secret key. The crypto_cntxt parameter is used for this purpose. The application can invoke other public key commands operating on the newly created public key by passing it as a parameter to the desired operation.

30

All public keysets are returned to the application in a packed form. Packed keys are defined as key structures in which the least

significant byte of the next public key structure member abuts the most significant byte of the current member. In this way, fragmentation within data structures is minimized and portability of data is enhanced.

Depending on the type of key being generated, some specific key generation parameters may be required. The sixth element of the kernel block's parameter area is used for this purpose:

- For DSA keys, the application may specify a seed key to be used for prime number generation. The seed key consists of a 16-bit counter and a 160-bit seed value. If no seed is specified (i.e., NULL is passed as this parameter), then the kernel will generate its own seed key. If a seed key is presented, the operation may fail if it cannot generate a prime from the seed. If so, the application should continue re-seeding the seed value and retrying until the operation succeeds. Upon success, the kernel will set the 16-bit counter.

This command is responsible for generating the random numbers and prime numbers to create the vectors for the public keyset. Furthermore, the command is responsible for executing the primality test specified by Rabin-Miller to accept/reject prime numbers. In all cases, the application is responsible for specifying the number of Rabin-Miller tests to perform. This allows the application to trade the strength of the generated prime numbers against the amount of time required to generate the primes.

The method parameter is a bit-mask controlling: 1) the type of prime numbers to generate, either weak or strong, and 2) how to search for prime numbers, either randomly (i.e. choosing a new random number for each prime number candidate) or sequentially from a random base (i.e. choose a random base and continue sequentially for each prime number candidate).

This command can be preempted by any command with the exception of the other public key commands.

Command Interface:

```

5      /* generate a Diffie-Hellman Public Keyset */
      cgx_gen_dh_pubkey(kernelblock    *kb,
                        unsigned short modulus_len,
                        unsigned short tests,
                        unsigned short method,
10     publickey        *pk,
        crypto_cntxt    *cc)

      /* generate a RSA Public Keyset */
      cgx_gen_rsa_pubkey(kernelblock    *kb,
15     unsigned short modulus_len,
        unsigned short base,
        unsigned short tests,
        unsigned short method,
        publickey      *pk,
20     crypto_cntxt     *cc)

      /* generate a DSA Public Keyset */
      cgx_gen_dsa_pubkey(kernelblock    *kb,
25     unsigned short modulus_len,
        unsigned short tests,
        unsigned short method,
        seedkey        *sk,
        publickey      *pk,
30     crypto_cntxt     *cc)

Arguments:
      kb->cb->cmd          = CGX_GEN_PUBKEY;

      cgx_gen_dh_pubkey(k, modulus_len, tests, method, pk, cc)
35     /* a pointer to the public key to work with */
      kb->cb->argument[0] = (VPTR)pk;
      /* a pointer to the public key's crypto_cntxt (for private
        * portion)
        */
40     kb->cb->argument[1] = (VPTR)cc;
      /* specify the public keyset to generate */
      kb->cb->argument[2] = (VPTR)CGX_DH_A;
      /* the length of the modulus key to generate, between 512 and
        * 2048 bits in increments of 64 bits, specify in units of bits
45     */

```

```

kb->cb->argument[3] = (VPTR) modulus_len;
/* the number of Rabin-Miller Primality Tests to perform */
kb->cb->argument[4] = (VPTR)tests;
/* the methods for generating prime numbers */
5 kb->cb->argument[5] = (VPTR)method;

cgx_gen_rsa_pubkey(k, modulus_len, base, tests, method, pk, cc)
/* a pointer to the public key to work with */
kb->cb->argument[0] = (VPTR)pk;
10 /* a pointer to the public key's crypto_cntxt (for private
    * portion)
    */
kb->cb->argument[1] = (VPTR)cc;
/* specify the public keyset to generate */
15 kb->cb->argument[2] = (VPTR)CGX_RSA_M;
/* the length of the modulus key to generate, between 512 and
    * 2048 bits in increments of 64 bits, specify in units of bits
    */
kb->cb->argument[3] = (VPTR) modulus_len;
20 /* the number of Rabin-Miller Primality Tests to perform */
kb->cb->argument[4] = (VPTR)tests;
/* the methods for generating prime numbers */
kb->cb->argument[5] = (VPTR)method;
/* the sixth parameter holds the starting point for the public
25 * exponent search.
    */
kb->cb->argument[6] = (VPTR) base;

cgx_gen_dsa_pubkey(k, modulus_len, tests, method, sk, pk, cc)
30
/* a pointer to the public key to work with */
kb->cb->argument[0] = (VPTR)pk;
/* a pointer to the public key's crypto_cntxt (for private
    * portion)
    */
35 kb->cb->argument[1] = (VPTR)cc;
/* specify the public keyset to generate */
kb->cb->argument[2] = (VPTR)CGX_DSA_M;
/* the length of the modulus key to generate, between 512 and
    * 2048 bits in increments of 64 bits, specify in units of bits
    */
40 kb->cb->argument[3] = (VPTR) modulus_len;
/* the number of Rabin-Miller Primality Tests to perform */
kb->cb->argument[4] = (VPTR)tests;
/* the methods for generating prime numbers */
45 kb->cb->argument[5] = (VPTR)method;
/* a pointer to the seed key */
kb->cb->argument[6] = (VPTR) NULL;

50 Status:
kb->sb->status = CGX_SUCCESS_S,

```

```

CGX_BAD_MODE_S,    /* invalid type requested */
CGX_INVALID_LEN_S, or /* invalid modulus or private
                      * length used
                      */
CGX_FAIL_S         /* failure during specific key
                      * generation operation
                      */

```

10 **See Also:** CGX_GEN_NEWPUBKEY, CGX_PUBKEY_ENCRYPT,
 CGX_PUBKEY_DECRYPT

GEN NEWPUBKEY (Generate Part of A Public Keyset)

15 **Command Name:** CGX_GEN_NEWPUBKEY

Command Description:

The Generate New public key operation is used to generate new public and private blocks for a Diffie-Hellman or DSA public keyset. This command is only valid for Diffie-Hellman or DSA public keysets.

20

The command allows the flexibility to import a modulus block from the application and use it to generate the new private and public blocks. Furthermore, the application has control over which parts to generate and return via the two control constants `CGX_X_V` (i.e. the private part) and `CGX_Y_V` (i.e. the public part). Using combinations of these control masks allows the application with a flexible key generation interface. The following uses of the masks are permissible:

- `CGX_X_V`
generates and returns a new private part, no public part
is returned.
- `CGX_Y_V`

generates and returns a new public part, no public part is returned. A private member must be passed in to create the new public part.

- CGX_X_V | CGX_Y_V

5 generates and returns a new private/public part, both parts are returned.

10 Once generated, the public key is returned to the application for use with subsequent public key commands. The modulus and public portions of the key are returned in the RED while the private portion of the key is returned in the BLACK, reflecting the sensitivity of the private data. As with all public key commands, this operation will generate key components which are stored in little endian order and are stored "packed".

15 For Diffie-Hellman keys, this operation provides the application with the ability to substitute its own value for the generator, eg. When a Diffie-Hellman key is generated, the value 2 is used for the generator.

20 Typically, this command is used to generate new public and private keys reusing the publicly shared modulus vectors so that a new secret keys can be derived. For Diffie-Hellman keys, the application can follow this command with the CGX_GEN_NEGKEY to generate the derived secret key from the receiver's public key.

25 For DSA keys, this operation can be followed by the CGX_SIGN and CGX_VERIFY operations using the newly generated DSA keyset.

This command can be preempted by any command which is not a public key command.

Command Interface:

```
5  /* modify the public and private vectors of a Diffie-Hellman or DSA Public Keyset */
   cgx_gen_newpubkey(kernelblock    *kb,
                      publickey     *modulus,
                      crypto_cntxt  *cc,
                      unsigned short key_gen)
```

Arguments:

```
10 kb->cb->cmd      = CGX_GEN_NEWPUBKEY;

   /* modulus must point to either a Diffie-Hellman or DSA keyset. */
15  /* the kernel will load the modulus portion of the key and use */
   /* it to create the new private and public portions of the */
   /* keyset. For Diffie-Hellman keys, the length member of the */
   /* private portion of the key will be used when generating the */
   /* private portion of the key. */
20 kb->cb->argument[0] = (VPTR) modulus;
   /* the crypto_cntxt to use when covering the newly generated */
   /* private portion of the keyset. */
   kb->cb->argument[1] = (VPTR) cc;
   /* specifies which key parts to return: CGX_X_V and/or CGX_Y_V */
25 kb->cb->argument[2] = (VPTR) key_gen;
```

Status:

```
30 kb->sb->status = CGX_SUCCESS_S,
                      CGX_BAD_KEYSET_S, /* keyset contains NULL data */
                      CGX_INVALID_LEN_S, /* invalid mod/priv len used */
                      CGX_FAIL_S, or /* invalid keyset type (RSA) or
                                      * error during key generation */
                      CGX_BAD_MODE_S, /* unrecognized keyset type */
```

See Also: CGX_GEN_PUBKEY, CGX_GEN_NEGKEY, CGX_SIGN, and CGX_VERIFY

40 GEN NEGKEY (Generate the DH Derived Secret Key)

Command Name: CGX_GEN_NEGKEY

GEN NEGKEY_GXY

(Cover and return the DH Derived Secret Key, $(g^x)^y$)

Command Name: CGX_GEN_NEGKEY_GXY

5

Command Descriptions:

These operations will complete the Diffie-Hellman exchange by deriving the shared secret key from the receiver's public key. The CryptIC supports dynamically negotiated keys as specified in the X9.42 Standard. Currently, the MQV 1 and 2 protocols are not supported. This command is only used for Diffie-Hellman public keysets.

To calculate the derived secret key the application must import the receiver's public key vector using the publickey definition. For the received public key, the application only needs to populate the public key field and algorithm type of the publickey. For the local private key, both the modulus and private keys are required. Because the private portion of the locally stored key is used, the crypto_cntxt which was used to cover the key must be presented as a parameter also. The secure kernel uses the imported public key to derive g^{xy} .

The commands CGX_GEN_NEGKEY and CGX_GEN_NEGKEY_GXY differ in what they do with the derived Diffie-Hellman shared secret key, g^{xy} .

The command CGX_GEN_NEGKEY truncates the shared secret key into a secret key of the desired length. The imported public key is discarded after the command completes. If for some reason the application needs to regenerate the derived secret key, it must reload the other parties' public key.

The application calling command CGX_GEN_NEGKEY must provide a destination key cache register to receive the derived secret

key and a destination black secret key and a KEKcc (a crypto context.) Furthermore, the application must specify the size of the derived secret key. The size can be specified as: 32-168 bits in length, specified in units of bytes (8 bits). The secret key will be created from the least
5 significant bits of the shared secret. By definition, secret keys created via this command are untrusted keys. The secret key can be either a key encryption key (KEK or KKEK) or a data encryption key (K). The KEKcc designates a kcr holding an encryption key and also supplies an initialization vector, which, using the kcr, is used to cover the derived
10 private key. The covered (black) key is then returned to the calling application via the argument supplied.

The command CGX_GEN_NEGKEY_GXY returns a covered (black) version of the shared secret key. The application calling command CGX_GEN_NEGKEY_GXY must provide a destination
15 public key and a KEKcc (a crypto context.) The KEKcc must designate a key cache register ID (kcr) to cover the shared key. The KEKcc supplies the initialization vector, which, using the kcr, is used to cover the shared secret key, g^{xy} . The covered key is then returned to the calling application via the argument supplied. Only the privatekey
20 member of the destination public key is populated; the calling application must ensure that the privatekey member of the destination public key has sufficient memory to receive the covered private key, viz., local public key's modulus size plus 8 bytes. The returned covered key may be used subsequently as input to the command
25 CGX_PRF_GXY or other CGX commands which facilitate IPsec operations.

Both commands can be preempted by any command which is not a public key command.

Command Interface:

```

/* derive a secret key using the Diffie-Hellman algorithm and public keyset */
cgx_negkey(kernelblock    *kb,
5         publickey       *localpk,
         crypto_cntxt     *lkek_cc,
         publickey       *remotepk,
         kcr              destkcr,
         UINT16           type,
         UINT16           len,
10        UINT16           use,
         secretkey        *bk,
         crypto_cntxt     *bkek_cc)

```

Arguments:

```

15      kb->cb->cmd          = CGX_GEN_NEGKEY;

/* a pointer to the publickey used to house the local public */
/* key. The modulus and private key fields are used to derive */
/* the shared secret. */
20      kb->cb->argument[0] = (VPTR)localpk;
/* a pointer to the crypto_cntxt which will be used to uncover */
/* the private portion of the local public key. */
kb->cb->argument[1] = (VPTR)lkek_cc;
/* a pointer to the publickey where the receiver's public key */
25 /* is stored. The receiver's public key is used to derive */
/* a secret key. All other fields except the pubkey member of */
/* the publickey are ignored. */
kb->cb->argument[2] = (VPTR)remotepk;
/* destination KCR ID for newly derived secret key */
30 kb->cb->argument[3] = (VPTR)destkcr;
/* the algorithm type of the secret key which will be derived */
kb->cb->argument[4] = (VPTR)type;
/* the length of the secret key which will be derived. The */
/* length indicates how many bits to use for the secret key, */
35 /* ranging from 32-168 bits (expressed in units of bytes). */
kb->cb->argument[5] = (VPTR)len;
/* how this key will be used. By definition, the key will be */
/* untrusted, but the application must identify this as either */
/* a key encryption key (KEK) or a data encryption key (K). */
40 kb->cb->argument[6] = (VPTR)use;
/* pointer to storage for the black version of the generated */
/* key. A NULL value for this parameter will cause a "dangling" */
/* key to be generated in a kcr within the key hierarchy. */
kb->cb->argument[7] = (VPTR)bk;
45 /* the crypto_cntxt used to cover the black version of the key. */
kb->cb->argument[8] = (VPTR)bkek_cc;
kb->cb->argument[9] = (VPTR)NULL;

```

Status:

```

kb->sb->status = CGX_SUCCESS_S,
                CGX_BAD_KEYSET_S, /* either local or remote */
                                   /* publickey is not a DH key */
                CGX_INVALID_KEY_LEN_S, /* invalid secret key len */
5         CGX_WEAK_KEY_S, /* remote public key was +/-1 */
                CGX_FAIL_S, or /* use or type not provided */
                CGX_INVALID_REG_S, /* invalid destkcr or bkek_cc */
                                   /* kcr */

```

Command Interface:

/* derive a secret key using the Diffie-Hellman algorithm and public keyset */

```

cgx_negkey_gxy(kernelblock *kb,
               publickey *localpk,
15      crypto_cntxt *lkek_cc,
               publickey *remotepk,
               publickey *gxy,
               crypto_cntxt *gyx_kek_cc)

```

Arguments:

```

kb->cb->cmd      = CGX_GEN_NEGKEY;

```

/* a pointer to the publickey used to house the local public */
/* key. The modulus and private key fields are used to derive */
/* the shared secret. */

```

kb->cb->argument[0] = (VPTR)localpk;
/* a pointer to the crypto_cntxt which will be used to uncover */
/* the private portion of the local public key. */

```

```

kb->cb->argument[1] = (VPTR)lkek_cc;
/* a pointer to the publickey where the receiver's public key */
/* is stored. The receiver's public key is used to derive */
/* a secret key. All other fields except the pubkey member of */
/* the publickey are ignored. */

```

```

kb->cb->argument[2] = (VPTR)remotepk;
kb->cb->argument[7] = (VPTR)NULL;
/* ptr to returned covered public key structure */
/* the crypto_cntxt used to cover the black version of the key. */

```

```

kb->cb->argument[8] = (VPTR)gyx_kek_cc;
kb->cb->argument[9] = (VPTR)gxy;

```

Status:

```

kb->sb->status = CGX_SUCCESS_S,
                CGX_BAD_KEYSET_S, /* either local or remote */
                                   /* publickey is not a DH key */
45      CGX_INVALID_KEY_LEN_S, /* invalid secret key len */
                CGX_WEAK_KEY_S, /* remote public key was +/-1 */
                CGX_FAIL_S, or /* use or type not provided */
                CGX_INVALID_REG_S, /* invalid destkcr or bkek_cc */
                                   /* kcr */

```

See Also: CGX_PRF_GXY, CGX_GEN_PUBKEY and
CGX_GEN_NEWPUBKEY

5

PUBKEY ENCRYPT (Encrypt Data Using RSA Public Key)

Command Name: CGX_PUBKEY_ENCRYPT

Command Description:

10 The Public Key Encrypt command is used to encrypt the
application's data using the RSA encryption algorithm. This operation
implements encryption or the RSA signature operation using the
pubkey member of a publickey structure. Control over which operation
is performed lies with the application. If the private key member if
NULL, RSA encryption occurs. If the public key is NULL, an RSA
15 signature is generated. The public keyset member of the RSA
publickey structure must be present for either of these operations to
occur.

20 The input data buffer, datain, must be a multiple of the length of
public keyset's modulus. Also, the value of the message must not
exceed the value of the modulus. (The message to be encrypted, like the
public keyset components, is stored in little endian order.)
Furthermore, the output buffer, dataout, must be at least as large as the
input buffer. Data from the datain buffer will be encrypted in blocks
equal to the size of the modulus and stored at the same relative offset in
25 the dataout buffer. This implies that the encryption algorithm will break
the input message into chunks of modulus length size and process each
chunk until the input is consumed.

If the output buffer isn't the same size as the input buffer, the application takes the chance of the operation failing if the result of the encryption can overflow the size of the output data buffer.

Command Interface:

```

5      /*encrypt the application's data using the RSA public keyset */
      cgx_pubkey_encrypt(kernelblock *kb,
                          publickey *pk,
                          crypto_cntxt *pkek_cc,
10     UINT16    dataoutpg,
                          BYTE    *dataout,
                          UINT16    datainpg,
                          BYTE    *datain,
                          UINT16    len)

```

Arguments:

```

15     kb->cb->cmd      = CGX_PUBKEY_ENCRYPT;

      /* a pointer to the public keyset to use for the encryption */
      /* operation. If the pkek_cc is NULL, the public portion of */
20     /* the keyset will be used for this operation. If the pkek_cc
      /* is not NULL, the private exponent will be used for encryption */
      bk->cb->argument[0] = (VPTR)(pk);
      /* a pointer to the crypto context used to cover the private */
      /* portion of the public keyset */
25     bk->cb->argument[1] = (VPTR)(pkek_cc);
      /* data page of output data buffer store the cipher-text */
      bk->cb->argument[2] = (VPTR)(dpg);
      /* output data buffer to store cipher-text of the message */
      bk->cb->argument[3] = (VPTR)(dp);
30     /* data page of input data buffer to be encrypted */
      bk->cb->argument[4] = (VPTR)(sdp);
      /* input data buffer to be encrypted */
      bk->cb->argument[5] = (VPTR)(sp);
      /* the length of the data to encrypt. The buffer length must be */
35     /* a multiple of the modulus size. In addition, the output */
      /* buffer length should be at least as large as the input buffer */
      bk->cb->argument[6] = (VPTR)(len);

```

Status:

```

40     kb->sb->status = CGX_SUCCESS_S,
                          CGX_BAD_KEYSET_S, /* both privkey and pubkey
                                          * present
                                          */
45     CGX_INVALID_LEN_S, /* invalid length specified */
                          CGX_BAD_KEYSET_S, /* keyset not an RSA keyset */
                          CGX_BAD_MODULUS_S, or /* message > modulus */

```

CGX_FAIL_S /* zero modulus */

See Also: CGX_GEN_PUBKEY, CGX_PUBKEY_DECRYPT

5 **PUBKEY DECRYPT (Decrypt Data Using RSA Public Key)**

Command Name: CGX_PUBKEY_DECRYPT

Command Description:

10 The Public Key Decrypt command is used to decrypt the application's data using the RSA decryption algorithm, or to verify an RSA signature. Control over which operation is performed lies with the application. If the public key member of the key is NULL, an RSA decryption will be performed. If the private key member of the structure is NULL, a signature verification will be performed. The public keyset must be provided by the application and contain the private key portion
15 of the keyset in order for this command to complete.

The size of the input data buffer, datain, must be a multiple of the public keyset's modulus size. Furthermore, the output buffer, dataout, must have a length greater than or equal to the size of the input buffer. The input buffer will be decrypted in chunks equal to the
20 size of the modulus, until the input buffer is consumed. The output message will be written into the dataout buffer at the same relative offset as the input buffer.

If the output buffer size isn't at least as large as the input buffer, the application takes the chance of the operation failing if the result of the decryption can overflow the size of the output data buffer.
25

Command Interface:

```
/*decrypt the application's data using the RSA public keyset */
cgx_pubkey_decrypt(kernelblock *kb,
publickey *pk,
crypto_cntxt *pkek_cc,
```

30


```

5          UINT16    dataoutpg,
          BYTE      *dataout,
          UINT16    datainpg,
          BYTE      *datain,
          UINT16    len)

```

Arguments:

```

kb->cb->cmd      = CGX_PUBKEY_DECRYPT;

10          /* a pointer to the public keyset to use for the decryption */
          /* operation. If the pkek_cc is NULL, the public portion of */
          /* the keyset will be used for this operation. If the pkek_cc
          /* is not NULL, the private exponent will be used for decryption */
          bk->cb->argument[0] = (VPTR)(pk);
15          /* a pointer to the crypto context used to cover the private */
          /* portion of the public keyset */
          bk->cb->argument[1] = (VPTR)(pkek_cc);
          /* data page of output data buffer store the plain-text */
          bk->cb->argument[2] = (VPTR)(dpg);
20          /* output data buffer to store plain-text of the message */
          bk->cb->argument[3] = (VPTR)(dp);
          /* data page of input data buffer to be decrypted */
          bk->cb->argument[4] = (VPTR)(sdp);
          /* input data buffer to be decrypted */
25          bk->cb->argument[5] = (VPTR)(sp);
          /* the length of the data to decrypt. The buffer length must be */
          /* a multiple of the modulus size. In addition, the output */
          /* buffer length should be at least as large as the input buffer */
          bk->cb->argument[6] = (VPTR)(len);
30

```

Status:

```

kb->sb->status = CGX_SUCCESS_S,
               CGX_BAD_OUTPUT, /* output buffer too small */
               CGX_BAD_INPUT, or /* input buffer too large */
               CGX_BAD_KEYSET_S /* empty key RAM */
35

```

See Also: CGX_GEN_PUBKEY and CGX_ENCRYPT_PUBKEY

EXPORT PUBKEY (Export An IRE Public Key)

Command Name: CGX_EXPORT_PUBKEY

Command Description:

5 The Export Pubkey command allows the application to move an IRE public key form into an external public key form. The external public form must be covered with a secret key, this is specified by the application via the command arguments.

10 The application must present a BLACK copy of the IRE public key to export along with the crypto_cntxt object to reference the KEK to uncover it. Also, the application must provide a buffer (i.e. ebk) to copy the external public key into that is converted from the BLACK IRE public key along with the publickey object. Furthermore, the exported public key can be covered by an untrusted key (i.e. K) or a KEK.

15 The IRE public key to export must not reside (or covered by) under a parent KEK that is trusted (i.e. LSV, GKEK, or trusted KEK). The main purpose of this command is to provide some sort of key interoperability between an IRE crypto device and some other vendor's crypto equipment (software or hardware based).

20 The public key to export is passed in via the argument, pk, the publickey object. The privkey, type and length members must be present in order to export the public key. Therefore, the private key is the only portion of the public keyset that is exported. The application can move the public and modulus portion itself.

25 As part of the command the application is allowed to program salt bits that will be prepended to the private key bits of the external

public key. When exporting a private key under a secret key the application has the choice of providing salt bits in multiples of 2 bytes, requesting the secure Kernel generate salt bits in multiples of 2 bytes, or not storing any salt bits. However, the total of salt bytes, data bytes, and the private key bytes must be a multiple of 8 bytes.

Also, the command allows the application to program data bits that will be appended to the private key bits of the external secret key. When exporting a private key under a secret key the application has the choice of providing data bits in multiples of 2 bytes, requesting the secure Kernel generate data bits in multiples of 2 bytes, or not storing any data bits. However, the total of salt bytes, data bytes, and the key bytes must be a multiple of 8 bytes.

If the argument, salt_len, is set to 0 the application requests that no salt bits be prepended. If the argument, salt, is NULL it requests that the secure Kernel generate salt_len bytes; otherwise the application is providing salt_len bytes of salt.

If the argument, data_len, is set to 0 the application requests that no data bits be prepended. If the argument, data, is NULL it requests that the secure Kernel generate data_len bytes; otherwise the application is providing data_len bytes of data.

Command Interfaces:

```
/*export an IRE private key under an untrusted secret key KEK */
cgx_export_pubkey( kernelblock  *kb,
    publickey    *pk,
    crypto_cntxt *pkek_cc,
    unsigned short *ebk,
    crypto_cntxt *ekek_cc,
    unsigned short *salt,
    unsigned short salt_len,
    unsigned short *data,
```

unsigned short data_len)

Arguments:

```

5      kb->cb->cmd      = CGX_EXPORT_PUBKEY;

      /* the BLACK IRE private key to export */
      kb->cb->argument[0] = (VPTR)pk;
      /* the KCR KEK location to uncover the BLACK IRE private key */
10     kb->cb->argument[1] = (VPTR)pkek_cc;
      /* the buffer to house the external private key */
      kb->cb->argument[2] = (VPTR)ebk;
      /* the KCR K or KEK location to cover the IRE private key */
      kb->cb->argument[3] = (VPTR)ekek_cc;
      /*salt bits to prepend to key bits */
15     kb->cb->argument[4] = (VPTR)salt;
      /* the number of salt bits to prepend to the key bits */
      /* this must be in units of 2bytes */
      kb->cb->argument[5] = (VPTR)salt_len;
      /*data bits to prepend to key bits */
20     kb->cb->argument[6] = (VPTR)data;
      /* the number of data bits to prepend to the key bits */
      /* this must be in units of 2bytes */
      kb->cb->argument[7] = (VPTR)data_len;

```

25

Status:

```

      kb->sb->status = CGX_SUCCESS_S, or
                     CGX_INVALID_REG_S

```

30

See Also: CGX_GEN_PUBKEY, CGX_GEN_NEWPUBKEY, and
CGX_IMPORT_PUBKEY

IMPORT PUBKEY (Import An IRE Public Key)

Command Name: CGX_IMPORT_PUBKEY

Command Description:

35

The Import Pubkey command allows the application to move an external public key form into an IRE internal form. The external public form must be covered with a secret key, this is specified by the application via the command arguments. The main purpose of this command is to provide some sort of key interoperability between an

IRE crypto device and some other vendor's crypto equipment (software or hardware based).

The application must present a BLACK copy of the external public key via the ibuf argument to import along with the crypto_cntxt object to reference the untrusted K or KEK to uncover it, an untrusted secret key. Also, the application must provide a public key object (i.e. pk) to convert the external public key buffer, ibuf, into an internal IRE public key form.

The external public key to import can not reside (or covered by) under a parent KEK that is trusted (i.e. LSV, GKEK, or trusted KEK). When it was exported it must have been under an untrusted secret key KEK only. If an attempt to uncover the imported key under anything but the correct untrusted KEK the operation will fail. However, the KEK to cover the converted public key with (the newly converted IRE public key) can be a trusted or untrusted secret key KEK.

The public key to imported is eventually copied into the argument, pk, the publickey object. The privkey, type and length members must be present in order to import the public key. Therefore, the private key is the only portion of the public keyset that is imported. The application can move the public and modulus portion itself.

As part of the command the application must indicate the number of salt bits that will be skipped before it can extract the private key bits. This is accomplished via the argument, skip_len, and it must be in multiples of 2 bytes.

Command Interfaces:

```
/*import an external private key under an untrusted secret key KEK */
cgx_import_pubkey( kernelblock  *kb,
```

```

        unsigned short *ibuf,
        crypto_cntxt *ikek,
        unsigned short skip_len,
        publickey *pk,
5      crypto_cntxt *pkek)

```

Arguments:

```

        kb->cb->cmd      = CGX_IMPORT_PUBKEY;

10      /* the BLACK IRE private key to import into */
        kb->cb->argument[0] = (VPTR)pk;
        /* the KCR K or KEK location to uncover the external private key */
        kb->cb->argument[1] = (VPTR)ikek;
        /* the buffer that houses the external private key */
15      kb->cb->argument[2] = (VPTR)ibuf;
        /* the KCR KEK location to cover the IRE private key */
        kb->cb->argument[3] = (VPTR)pkek;
        /*salt bits to skip */
        kb->cb->argument[5] = (VPTR)skip_len;
20

```

Status:

```

        kb->sb->status = CGX_SUCCESS_S, or
                        CGX_INVALID_REG_S

25      See Also: CGX_GEN_PUBKEY, CGX_GEN_NEWPUBKEY, and
                CGX_EXPORT_PUBKEY

```

Digital Signature Commands**SIGN (Digitally Sign A Message)**

```

30      Command Name:      CGX_SIGN

```

Command Description:

The digital signature sign command is used to sign the application's message using the DSA digital signature algorithm.

The application can pass in a covered(i.e. BLACK) DSA public key or an uncovered (i.e. RED) public key. If the color of the public key is determined by the argument, kek. If it is NULL then the public key is assumed to be in the RED form. If the argument is non-NULL

then the public key is in the BLACK form and must be uncovered with the kek.

To digitally sign a message a one-way hash result is processed by the digital signature algorithm. The result of the sign command is placed into the signblock.

The result is returned in the signblock, it's the data in this block that must be used by the CGX_VERIFY command to verify the message's signature.

Also, the application is allowed to pass in K, of the DSA vectors. Remember K is typically a random vector that should be considered secret. If K is known by anyone they can forge the signature. Therefore, using this feature must be used only in extreme cases; otherwise pass in a NULL pointer and let the secure Kernel generate K.

Prior to using this command the application must calculate the one-way hash value of the message to sign. The result of the one-way hash value is stored in the hash_cntxt block object. This block must be passed in as an argument to the command.

Command Interface:

```
/*DSA sign the application's message */
cgx_sign( kernelblock  *kb,
          publickey    *pk,
          crypto_cntxt *kek,
          signblock     *sb,
          hash_cntxt    *hc,
          unsigned short *K)
```

Arguments:

```
kb->cb->cmd = CGX_SIGN;
```

```

/* the DSA public keyset */
kb->cb->argument[0] = (VPTR)pk;
/* signature block, contains result of signature (input/output) */
kb->cb->argument[1] = (VPTR)s;
5  /* one-way hash value of message to sign */
kb->cb->argument[2] = (VPTR)hc;
/* The KEK to uncover the DSA private part, NULL private is RED */
kb->cb->argument[3] = (VPTR)kek;
10 /* vector K, NULL Kernel generates K, highly risky to use */
kb->cb->argument[4] = (VPTR)K;
/* a non-zero value requests the sign be verified */
kb->cb->argument[5] = (VPTR)recheck;

Status:
15 kb->sb->status = CGX_SUCCESS_S,
CGX_FAIL_S,
CGX_BAD_KEYSET_S /* empty key RAM or wrong alg */

```

20 **See Also:** CGX_VERIFY, CGX_INIT_HASH, and CGX_HASH_DATA

VERIFY (Verify A Digital Signature)

Command Name: CGX_VERIFY

Command Description:

25 The digital signature verify command is used to verify the signature of the application's message using the DSA public key algorithm.

To verify a digital signature a one-way hash is calculated over the message and its result is processed by the digital signature algorithm for verification. The result of the verify command is compared to the result already in the signblock.

30

The result of the calculation is compared to the signature result already stored in the signblock, if the results compare then the signature is valid; otherwise the signature fails and is not valid.

35 The application need only pass in the modulus and public parts of the DSA public keyset, the private part is ignored.

Prior to using this command the application must calculate the one-way hash value of the message to verify. The result of the one-way hash value is stored in the hash_cntxt block object. This block must be passed in as an argument to the command.

Command Interface:

```
/*DSA verify the application's signed message */
cgx_verify( kernelblock  *kb,
            publickey    *pk,
            signblock     *sb,
            hash_cntxt    *hc)
```

Arguments:

```
kb->cb->cmd = CGX_VERIFY;
```

```
/* the DSA public keyset */
kb->cb->argument[0] = (VPTR)pk;
/* signature block, contains result of signature (input/output) */
kb->cb->argument[1] = (VPTR)sb;
/* one-way hash value of message to sign */
kb->cb->argument[2] = (VPTR)hc;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S,
               CGX_INVALID_SIGNATURE_S, or /* sig not valid */
               CGX_BAD_KEYSET_S /* empty key RAM or wrong alg */
```

See Also: CGX_SIGN, CGX_INIT_HASH, and CGX_HASH_DATA

Extended Algorithm Commands

LOAD EXTENDED (Load/Enable Extended Algorithm Block)

Command Name: CGX_LOAD_EXTENDED

Command Description:

The load extended algorithm command allows up to 8 kilowords of expansion for secure kernel program memory. This block can be used to augment the secure kernel with algorithms (e.g., IDEA or Blowfish) or features which are not contained in the standard kernel.

Because this command will cause the kernel to accept the application code as legitimate, the application must be signed by IRE. When invoking the load extended command, the application must present the signature provided by IRE. The signature will be authenticated by the secure kernel before enabling the extended algorithm block. If the signature is not successfully authenticated, the secure kernel will not transfer the application code to the extended algorithm block and hardware protection for the extended algorithm block will not be enabled.

The extended algorithm block must be bound beginning at location `xxxx`. When the signature over the algorithm block has been verified, the kernel will transfer the application from program memory into the extended algorithm block and enable kernel protection over the block.

Once the extended algorithm block has been authenticated and loaded, the application can invoke operations in the extended algorithm block via the `CGX_EXEC_EXTENDED` command. When the application no longer needs the extended algorithm block, the block may be disabled via the `CGX_CLEAR_EXTENDED` command.

If the application does not use the extended algorithm block to expand the secure kernel, it may still be used for application code.

Command Interface:

```
/* load the extended algorithm block and enable hardware protection */
cgx_load_extended(kernelblock *kb,
                  unsigned char pm *prog,
                  UINT16 len,
                  signblock *sb)
```

Arguments:

```

kb->cb->cmd    = CGX_LOAD_EXTENDED;

/* the address of the extended algorithm block */
kb->cb->argument[0] = (VPTR)prog;
5  /* the length of the block in 24-bit words */
kb->cb->argument[1] = (VPTR)len;
/* the signature for the extended algorithm block */
kb->cb->argument[2] = (VPTR)sb;

10  Status:
kb->sb->status = CGX_SUCCESS_S,
               CGX_ACCESS_DENIED, /* Extended algorithms not
                                   * permitted.
                                   */
15  or CGX_INVALID_SIGNATURE_S.

```

See Also: CGX_EXEC_EXTENDED and CGX_CLEAR_EXTENDED

EXECUTE EXTENDED (Execute Extended Algorithm Block)

20 **Command Name:** CGX_EXEC_EXTENDED

Command Description:

The execute extended algorithm command allows the application to invoke operations contained in the extended algorithm block. Operations are permitted a maximum of 10 parameters stored in the *argument* member of the *cmdblock* structure. In order for a request to execute from the extended algorithm block to be honored, the algorithm must already have been loaded via the CGX_LOAD_EXTENDED command.

Before invoking the extended algorithm block, the secure kernel will verify that the block is active (i.e., has a valid algorithm loaded). If so, control is transferred to the extended algorithm block via the block's entry point (the specific location is yet to be determined). It is the algorithm block's responsibility to handle branching to the appropriate handler operation. In other words, once control is transferred to the extended algorithm block's entry point, the application is responsible for handling the command.

Upon completion of command processing, the extended algorithm can return a UINT16 value indicating the completion status of the command. This value will be available to the application in the *status* field of the kernel block.

5

Command Interface:

```
/* execute from the extended algorithm block */
cgx_execute_extended(kernelblock *kb)
```

10

Arguments:

```
kb->cb->cmd      = CGX_EXEC_EXTENDED;
```

```
/* arguments can be passed by the application via the argument
 * array. This data will be available to the algorithm block.
 */
```

15

Status:

```
kb->sb->status = CGX_ACCESS_DENIED_S, /* extended algorithms not
 * permitted.
 */
```

20

```
CGX_FAIL_S, /* extended algorithm block not
 * initialized.
 */
```

```
result from extended algorithm.
```

25

See Also: CGX_LOAD_EXTENDED and CGX_CLEAR_EXTENDED

Hash Commands

30

HASH INIT (Initialize The HASH Operator)

Command Name: CGX_HASH_INIT

Command Description:

The Initialize Hash command is used to initialize a Hash context block, hash_cntxt, and prepare it for the start of a Hash calculation. The application must invoke the initialize Hash operation prior to starting a new Hash calculation; if not the hash_cntxt is not initialized and the Hash calculation will be incorrect. Having the Initialize Hash command allows the application to run simultaneous Hash calculations,

35

interleaving the calls to CGX_HASH_DATA. This is facilitated due to the Hash context management.

The Hash operations support SHS-1 and MD5 one way Hash algorithms. Both Hash algorithms have a limit of 2^{64} bits message length and both process units of words.

The application must setup the hash_cntxt with an algorithm mode (i.e., CGX_SHS_A or CGX_MD5_A) prior to the execution of the initialize Hash command. Upon completion of this operation, the hash context will contain a NULL value in the digest member of the hash_cntxt. When the hash is closed, the digest member will be a valid pointer to the hash digest.

Command Interface:

```
/* initialize HASH context, prepare to HASH */
cgx_hash_init(kernelblock *kb,
               hash_cntxt *hb)
```

Arguments:

```
kb->cb->cmd = CGX_INIT_HASH;
```

```
/* the hash context block to initialize */
kb->cb->argument[0] = (VPTR)hb;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S, or
                CGX_BAD_MODE_S /* only SHS-1 or MD5 available */
```

See Also: CGX_HASH_DATA

HASH DATA (HASH Customer Data)

Command Name: CGX_HASH_DATA

Command Description:

5 The Hash Data command is used to calculate a Hash value of the application's data. The hash may include a data key either prepended to the data, appended to the data, or both (the keys need not be the same). The key may not be inserted into the middle of the data due to security concerns [in this context, "data" encompasses the *entire* data stream to be hashed – in other words, keys can only be hashed at the very beginning or end of the data]. Neither the key, nor its parent in the key hierarchy (i.e., its KEK) may be trusted. Any attempt to hash a trusted key will result in a failure. The hash_cntxt block allows the application to run simultaneous Hash calculations interleaving the calls to CGX_HASH_DATA.

10 The HASH operations support SHS-1 and MD5 one-way Hash algorithms. Both Hash algorithms have a message length limit of 2^{64} bits and operate on units of bytes.

15 The application must set up the hash_cntxt with an algorithm mode (i.e., CGX_SHS_M or CGX_MD5_M) prior to the execution of the HASH data command. The application must set the context block up and invoke the CGX_INIT_HASH command first.

20 Because it is illegal to request hashing of the LSV, a value of non-0 for the key indicates that a keyed hash is being requested. When keyed hashing is requested, the position of the key in the data must be identified as either CGX_PREPEND_O, CGX_APPEND_O, or both. The *position* parameter is a bit field.

25 Setting the *final* argument to 1 signals the end of the message. This causes the secure kernel to prepare the message digest. If the *final* operation is not performed, the message digest is incorrect. If the application has no message to Hash and has set *final* to 1 then it must set message_len to 0. When message_len is set to 0 the message buffer

is ignored. However, if there are still more messages to Hash, the application must set *final* to 0.

Command Interface:

```

5  /* calculate HASH of message */
   cgx_hash_data(kernelblock  *kb,
                 unsigned short message_pg,
                 unsigned short *message,
10  unsigned long  length,
                 unsigned short final,
                 kcr      key,
                 unsigned short position,
                 hash_cntxt  *hb)

```

Arguments:

```

15  kb->cb->cmd      = CGX_HASH_DATA;

   /* the hash context block */
   kb->cb->argument[0] = (VPTR)hb;
20  /* the data page of the message. If necessary, the underlying
   * software will up temporarily switch pages in order to access
   * the message data.
   */
   kb->cb->argument[1] = (VPTR)message_pg;
25  /* pointer to length bytes of memory to HASH, a message can be
   * variable length. The application can make as many calls to
   * calculate the HASH message digest. This is because of the
   * hash_cntxt block. If final operation and no message set to NULL.
   */
30  kb->cb->argument[2] = (VPTR)message;
   /* length, set to 0 if no more data to hash. The data length is 4
   * bytes long, broken into two arguments. Length represents the
   * number of bytes to hash, up to a total of 264 bits (261
   * bytes) to hash.
35  */
   kb->cb->argument[3] = (VPTR)((length >> 16) & 0xffff);
   kb->cb->argument[4] = (VPTR)(length & 0xffff);
   /* if no more message data to hash, set final to 1 */
   /* otherwise there is more data to come or at least another */
40  /* call to CGX_HASH_DATA so set final to a 0 */
   kb->cb->argument[5] = (VPTR)final;
   /* identify the KCR containing the key to be hashed into the */
   /* data. The key (and its KEK) cannot be trusted keys or the*/
   /* operation will fail */
45  kb->cb->argument[6] = (VPTR)key;
   /* specify where the key is to be added relative to the data */
   /* this parameter is a bit-field; so, prepending and appending */
   /* of the key is permitted using arithmetic OR */

```

```
kb->cb->argument[7] = (VPTR)position;
```

Status:

```
kb->sb->status = CGX_SUCCESS_S,  
5 CGX_FAIL_S, /* length = 0 and final = 0 */ or  
CGX_BAD_MODE_S /* only SHS-1 or MD5 available */
```

See Also: CGX_INIT_HASH

10 HASH ENCRYPT (Hash and Encrypt User Data)

Command Name: CGX_HASH_ENCRYPT

Command Description:

15 The Hash and Encrypt command is used to perform a hash calculation and symmetrical encryption of the customer's data. Having a hash_cntxt and crypto_cntxt block allows the application to run simultaneous hash calculations interleaving the calls with CGX_HASH_DATA and CGX_HASH_DECRYPT commands; this is because of the context management objects, hash_cntxt and crypto_cntxt.

20 The hash algorithms supported are SHS-1 and MD5 one-way hash algorithms. For any invocation of hash_encrypt, the algorithms have a limit of 2^{16} bytes. Due to requirements imposed by the encryption operations, the length must be evenly divisible by 8 (the encryption/decryption operations operate on blocks of 8 bytes). A message not evenly divisible by 8 must be padded by the application. 25 The message or block may be padded with any pattern the application wishes.

30 The encrypt algorithms supported are: DES, Triple DES, and RC5, in several modes: ECB, CFB, OFB, and CBC. The encryption algorithms only support block encryption, a block must be divisible by 64 bits long (to be performed by the application). A encrypted data

session can extend beyond one call to the encryption command; this is accomplished via a `crypto_cntxt` block (described below).

The data buffers, `datain` (plain-text) and `dataout` (cipher-text), can share the same address.

5 Prior to invoking the hash and encrypt command the application must setup the `crypto_cntxt` block. The application is responsible for setting the configuration, masking in one of the special key load options (described below), setting the appropriate secret key KCR ID, and for priming the iv buffer with an initial random number (only for the first
10 call). After this, the application need not modify the `crypto_cntxt` block unless something changes (i.e., secret key KCR ID location, and/or the iv to cause a resynchronization).

15 The iv buffer of the `crypto_cntxt` block must be read- and write-able; the iv buffer is used to maintain the feed-back register for the CBC, CFB, and OFB modes. In the ECB mode the iv buffer is ignored.

20 The hash and encryption command allows the application to mask in one of the special mode control bits defined in section 0, into the config field of the `crypto_cntxt` block. The control bits are used by the secure kernel to determine how to load secret keys before the actual encryption of the plain-text takes place. The control bits allow the
25 application to request one of these options: auto-load, force-load, or no-load. Auto-load allows the secure kernel to check which key is currently loaded into the KG, if it's the same as the key specified for the encryption command it is not loaded; otherwise the key is loaded. Force-load tells the secure kernel to always load the key. No-load tells the secure kernel to not load the key; more than likely the application has already loaded the key (maybe via the `CGX_LOAD_KG` command). By default auto-load is assumed.

The application must also set up the `hash_cntxt` with an algorithm (i.e., `CGX_SHS_M` or `CGX_MD5_M`) prior to the execution of the `hash_data` command. The application must set-up the context block and invoke the `CGX_INIT_HASH` command first.

5 Setting the final argument to 1 instructs the secure kernel to prepare the hash value for return to the application. If the final operation is not performed the message digest is incorrect. If the application has no message to hash and has set final to 1 then it must set length to 0. When length is 0, the datain buffer is ignored. If there are
10 still more messages to hash the application must set final to 0.

 The *order* parameter allows the application to specify the hash and encrypt order. The value `CGX_HASH_CRYPT_O` indicates hash before encrypt and `CGX_CRYPT_HASH_O` implies encrypt before hash.

15 The *offset* parameter indicates the number of bytes to process according to the value of order before applying the other operation. That is, if *offset* == 4 and *order* == `CGX_HASH_CRYPT_O`, then 4 bytes of data will be hashed, and then the remaining length-4 bytes will be hashed and encrypted. The value of *offset* must be even (i.e., align
20 on a word boundary).

 The secret key to be loaded as referenced via the `crypto_cntxt` must have the key *usage* setting of `CGX_KCR_K`. This command only allows traffic or data keys to be loaded or it will fail. The key usage information is programmed by the application at the time a secret key
25 has been loaded, generated, derived, or negotiated and is maintained securely by the secure Kernel. This means an LSV, GKEK, or KEK type of secret key can not be loaded by this command.

Command Interface:

```

/* calculate HASH and encrypt the message */
cgx_hash_encrypt( kernelblock  *kb,
                  UINT16      datain_pg,
                  UINT16      *datain,
5                  UINT16      dataout_pg,
                  UINT16      *dataout,
                  UINT16      length,
                  UINT16      order,
10                 UINT16      offset,
                  UINT16      final,
                  crypto_cntxt *cb
                  hash_cntxt  *hb)

```

Arguments:

```

15         kb->cb->cmd          = CGX_HASH_ENCRYPT;

/* mode selection, mask the special mode and configuration
* control bits with the mode here via the crypto_cntxt block
*/
20         kb->cb->argument[0] = (VPTR)cb;
/* the hash context block to initialize */
kb->cb->argument[1] = (VPTR)hb;
/* input message to be encrypted and hashed (data page, offset) */
25         kb->cb->argument[2] = (VPTR)datain_pg;
kb->cb->argument[3] = (VPTR)datain;
/* number of bytes (byte = 8 bits) to encrypt */
kb->cb->argument[4] = (VPTR)length;
/* if no more message data, set final to a 1 */
/* otherwise there are more messages to come or at least another */
30         /* call to CGX_HASH_DATA so set final to a 0 */
kb->cb->argument[5] = (VPTR)final;
/* if hash before encrypt set order to a CGX_HASH_CRYPT_O */
/* if encrypt before hash set order to a CGX_CRYPT_HASH_O */
kb->cb->argument[6] = (VPTR)order;
35         /* output the encrypted data blocks (data page, offset) */
kb->cb->argument[7] = (VPTR)dataout;
kb->cb->argument[8] = (VPTR)dataout_pg;
/* hash/decrypt offset */
40         kb->cb->argument[9] = (VPTR)offset;

```

Status:

```

kb->sb->status = CGX_SUCCESS_S,
                CGX_WEAK_KEY_S,
                CGX_BAD_MODE_S,
45         CGX_EMPTY_REG_S, or
                CGX_INVALID_REG_S

```

See Also: CGX_INIT_HASH, CGX_ENCRYPT, and CGX_HASH_DATA

HASH DECRYPT (Hash and Decrypt User Data)**Command Name:** CGX_HASH_DECRYPT**Command Description:**

5 The hash and decrypt command is used to perform a hash calculation and symmetrical decryption of the customer's data. Having a hash_cntxt and crypto_cntxt blocks allows the application to run simultaneous hash calculations interleaving the calls with CGX_HASH_DATA and CGX_HASH_ENCRYPT commands; this is
10 because of the context management objects, hash_cntxt and crypto_cntxt.

 The one-way hash algorithms supported are: SHS-1 and MD5. For any invocation of hash_decrypt, the algorithms have a limit of 2^{16} bytes. Due to requirements of the encryption algorithms, message
15 lengths must be evenly divisible by 8. Any message or block not evenly divisible by 8 bytes must be padded by the application. The message may be padded with any pattern which the application chooses.

 The decrypt algorithms supported are: DES, Triple DES, and
20 RC5, in several modes: ECB, CFB, OFB, and CBC. The decryption algorithms only support block decryption, a block must be 64 bits long. The decryption command can handle up to 2^{13} bytes at one time. Furthermore, an encrypted data session can extend beyond one call to the decryption command; this is accomplished via a crypto_cntxt block
25 (described below).

 The data buffers, datain (plain-text) and dataout (cipher-text), can share the same address.

Prior to invoking the hash and decrypt command the application must setup the `crypto_cntxt` block. The application is responsible for setting the configuration, masking in one of the special key load options (described below), setting the appropriate secret key KCR ID, and for priming the iv buffer with an initial random number (only for the first call). After this, the application need not modify the `crypto_cntxt` block unless something changes (i.e., secret key KCR ID location, algorithm, and/or the iv to cause a resynchronization).

The iv buffer of the `crypto_cntxt` block must be read- and write-able; the iv buffer is used to maintain the feed-back register for the CBC, CFB, and OFB modes. In the ECB mode, the iv buffer is ignored.

The hash and decrypt command allows the application to mask one of the special mode control bits, defined in section 0, into the algorithm field of the `crypto_cntxt` block. The control bits are used by the secure kernel to determine how to load secret keys before the actual encryption of the plain-text takes place. The control bits allow the application to request one of these options: auto-load, force-load, or no-load. Auto-load allows the secure kernel to check which key is currently loaded into the KG, if its the same as the key specified for the decryption command it is not loaded; otherwise the key is loaded. Force-load tells the secure kernel to always load the key. No-load tells the secure kernel to not load the key; more than likely the application has already loaded the key (maybe via the `CGX_LOAD_KG` command). By default auto-load is assumed.

The application must also set up the `hash_cntxt` with an algorithm mode (i.e., `CGX_SHS_M` or `CGX_MD5_M`) prior to the

execution of the `hash_data` command. The application must setup the context block and invoke the `CGX_INIT_HASH` command first.

5 Setting the final argument to 1 signals the secure kernel to prepare the message digest for the application. This is important to correctly calculating the message digest. If the final operation is not performed the message digest is incorrect. If the application has no message to hash and has set final to 1. When `nblocks` is set to 0 the datain buffer is ignored. However, if there are still more messages to
10 hash the application must set final to 0.

 The parameter *order* allows the application to specify the hash and decrypt order. The value `CGX_HASH_CRYPT_O` indicates hash before decrypt and `CGX_CRYPT_HASH_O` implies decrypt before
15 hash. The *offset* parameter indicates the number of bytes to process according to the value of *order* before applying the other operation. That is, if *offset* == 4 and *order* == `CGX_HASH_CRYPT_O`, then 4 bytes of data will be hashed, and then the remaining length-4 bytes will be hashed and decrypted. The value of *offset* must be even (i.e., align
20 on a word boundary).

 The secret key to be loaded as referenced via the `crypto_cntxt` must have the *key usage* setting of `CGX_KCR_K`. This command only allows traffic or data keys to be loaded or it will fail. The key usage
25 information is programmed by the application at the time a secret key has been loaded, generated, derived, or negotiated and is maintained securely by the secure Kernel. This means an LSV, GKEK, or KEK type of secret key can not be loaded by this command.

Command Interface:

```

/* calculate HASH and encrypt the message */
cgx_hash_decrypt( kernelblock  *kb,
5          UINT16    datain_pg,
          UINT16    *datain,
          UINT16    dataout_pg,
          UINT16    *dataout,
          UINT16    length,
10         UINT16    order,
          UINT16    offset,
          UINT16    final,
          crypto_cntxt *cb
          hash_cntxt  *hb)

15  Arguments:
      kb->cb->cmd      = CGX_HASH_DECRYPT;

      /* algorithm and mode selection,
      * mask one of the special algorithm
20  * control bits with the mode here via the crypto_cntxt block
      */
      kb->cb->argument[0] = (VPTR)cb;
      /* the hash context block to initialize */
      kb->cb->argument[1] = (VPTR)hb;
25  /* data to be decrypted and hashed, data page and offset */
      kb->cb->argument[2] = (VPTR)datain_pg;
      kb->cb->argument[3] = (VPTR)datain;
      /* number of bytes to hash/decrypt */
      kb->cb->argument[4] = (VPTR)nblocks;
30  /* if no more data, set final to 1 */
      /* otherwise there is more data to come or at least another */
      /* call to CGX_HASH_DATA so set final to a 0 */
      kb->cb->argument[5] = (VPTR)final;
      /* if hash before decrypt set order to a CGX_HASH_CRYPT_O */
35  /* if decrypt before hash set order to a CGX_CRYPT_HASH_O */
      kb->cb->argument[6] = (VPTR)order;
      /* output the decrypted data blocks, data page and offset */
      kb->cb->argument[7] = (VPTR)dataout_pg;
      kb->cb->argument[8] = (VPTR)dataout;
40  /* hash/decrypt offset */
      kb->cb->argument[9] = (VPTR)offset;

Status:
      kb->sb->status = CGX_SUCCESS_S,
45                      CGX_WEAK_KEY_S,
                      CGX_BAD_MODE_S,
                      CGX_EMPTY_REG_S, or
                      CGX_INVALID_REG_S

```

See Also: CGX_INIT_HASH, CGX_DECRYPT, and CGX_HASH_DATA

Math Commands

5 **MATH (Math Utilities)**

Command Name: CGX_MATH

Command Description:

10 The Math utilities expose multiple primitive Math functions
which are accelerated by the Public Key engine. Primitive functions
include: 64-bit vector Add, 64-bit vector Subtract, 64-bit vector
Multiply, 64-bit vector Exponentiate, and 64-bit vector Reduction.

More information to follow in next revision.

Command Interface:

15 /* Information Coming */
cgx_math(kernelblock *kb,
 ...)

20 **Arguments:**
 kb->cb->cmd = CGX_MATH;

25 **Status:**
 kb->sb->status = CGX_SUCCESS_S,

PRF Commands

30 The commands in this section allow an application to perform manipulations on
keys and other secret data required to perform certain IPSec transforms. (“PRF” is an
abbreviation of pseudo-random function.)

MERGE KEY

Command Name: CGX_MERGE_KEY

Command Description:

5 The CGX_MERGE_KEY command takes key material from two secret keys and combines the material to form a third secret key. The key material in two input keys, key1 and key2, is combined in a caller specified way.

 The input keys are presented to the function in the black; crypto contexts for each must be supplied to uncover them internally. The resulting output key will be returned in the black so a crypto context for it must also be supplied as a function argument.

10 The output key pointer, bk, may point to the same memory as an input key pointer, in which case the output will overwrite the input. Also, any of the kek pointers supplied may be equal, in which case the same covering key will be used to cover or uncover parameters, as appropriate.

15 The possible combine operations are concatenate, exclusive-or, and hash. The resulting material (or the leading bytes of the resulting material, if the resulting material is more than needed to create the new key) becomes the key material for a new key.

20 For the concatenate operation, key2's key material is appended to that of key1, resulting in material whose length is the sum of the lengths of key1 and key2.

25 For the exclusive-or operation, key2's key material is bitwise exclusive-ored with that of key1, resulting in material whose length is the greater of the lengths of key1 and key2. (The shorter key is zero padded for purposes of the exclusive-or.)

For the hash operation, key2's key material is appended to that of key1, resulting in intermediate material whose length is the sum of the lengths of key1 and key2. Then the resulting material is hashed by the specified hash algorithm and the resulting hash digest is used as the new key's material. The length of the material in this case is the hash digest length.

For operations concatenate and hash, the order of input keys is significant, especially when one is attempting to perform operations according to an external specification. Another way to state which input order results in which intermediate key material order is that, for operations concatenate and hash, key2 becomes the most significant and key1 becomes the least significant part of the intermediate key material after concatenation.

The new key's length and type are specified as arguments. The supplied length must be in the allowed range for secret keys and the key material generated by combining the input keys must be sufficient to produce a key of the specified length.

The use of the new key may be KEK or K (encryption key.) The two input keys must either both be KEKs or neither be a KEK.

The astute reader will note that three or more input keys may be combined by merging the output of one merge_key operation with yet another input key, and repeating this step as often as necessary.

Command Interface:

```
/* merge two keys and create a third from the combined key material*/  
cgx_merge_key( kernelblock  *kb,  
               secretkey   *key1,  
               crypto_cntxt *kek1,  
               secretkey   *key2,
```

```

5      crypto_cntxt  *kek2,
      UINT16         operation,
      UINT16         type,
      UINT16         length,
      UINT16         use,
      secretkey      *bk,
      crypto_cntxt   *bk_kek)

```

Arguments:

```

10      kb->cb->cmd      = CGX_MERGE_KEY;
      kb->cb->argument[0] = (VPTR)key1;
      /* the first input key to be merged */

      kb->cb->argument[1] = (VPTR)kek1;
15      /* crypto context used to uncover key1 */

      kb->cb->argument[2] = (VPTR)key2;
      /* the second input key to be merged */

      kb->cb->argument[3] = (VPTR)kek2;
20      /* crypto context used to uncover key2 */

      kb->cb->argument[4] = (VPTR)operation;
      /* CGX_CONCAT_O, CGX_XOR_O, CGX_HASH_O | CGX_SHS_A,
25      CGX_HASH_O | CGX_MD5_A, or CGX_HASH_O | <other_hash_alg_type> */

      kb->cb->argument[5] = (VPTR)type;
      /* key type of result key to be produced */
      /* CGX_DES_A or CGX_TRIPLE_DES_A or other key type */

30      kb->cb->argument[6] = (VPTR)length;
      /* desired length in bytes of result key to be created */

      kb->cb->argument[7] = (VPTR)use;
35      /* intended use of result key: key type and trust attribute */

      kb->cb->argument[8] = (VPTR)bk;
      /* pointer to result key to be returned */

      kb->cb->argument[9] = (VPTR)bk_kek;
40      /* crypto context used to cover bk */

```

Status:

```

45      kb->status =
      CGX_SUCCESS_S,
      CGX_BAD_KEY_S,
      CGX_NULL_PTR_S,
      CGX_FAIL_S

```

MERGE LONG KEY

Command Name: CGX_MERGE_LONG_KEY

Command Description:

5 The CGX_MERGE_LONG_KEY command is quite similar to
the CGX_MERGE_KEY command. The essential difference is that the
output key created by CGX_MERGE_LONG_KEY is not a data
encryption key; rather it is merely a container for key material that can
be used subsequently (for example by command
CGX_EXTRACT_LONG) to create encryption keys. The output data
10 type of CGX_MERGE_LONG_KEY is a container, not a true key; it is
perhaps misnamed as a **longkey** data type. A variable of this type can
hold up to 64 bytes of key information. Such a data type provides
intermediate storage, for example, for the 48 bytes resulting from
concatenating two 24 byte keys, which then can be used (by
15 CGX_EXTRACT_LONG) to produce an encryption key from the
middle 24 bytes of the concatenation.

 The CGX_MERGE_LONG_KEY command takes key material
from two keys and combines the material to form a new long key. The
first input key, key1, may be either an ordinary encryption key (type
20 secretkey) or a longkey. The second input key, key2, must be an
ordinary encryption key. The key material in two input keys, key1 and
key2, is combined in a caller specified way.

 The input keys are presented to the function in the black; crypto
contexts for each must be supplied to uncover them internally. The
25 resulting long output key will be returned in the black so a crypto
context for it must also be supplied as a function argument.

 The output key pointer, lk, may point to the same memory as an
input key pointer, (provided the allocated memory is sufficient to hold a

longkey type datum) in which case the output will overwrite the input. Also, any of the kek pointers supplied may be equal, in which case the same covering key will be used to cover or uncover parameters, as appropriate.

5 The possible combine operations are concatenate, exclusive-or, or hash. The resulting material becomes the key material for the new key.

10 For the concatenate operation, key2's key material is appended to that of key1, resulting in material whose length is the sum of the lengths of key1 and key2.

 For the exclusive-or operation, key2's key material bitwise exclusive-ored with that of key1, resulting in material whose length is the greater of the lengths of key1 and key2. (The shorter key is zero padded for purposes of the xor.)

15 For the hash operation, key2's key material is appended to that of key1, resulting in material whose length is the sum of the lengths of key1 and key2. Then the resulting material is hashed by the specified hash algorithm and the resulting hash digest is used as the new key's material. The length of the material in this case is the hash digest length.

20 For operations concatenate and hash, the order of input keys is significant, especially when one is attempting to perform operations according to an external specification. Another way to state which input order results in which intermediate key material order is that, for operations concatenate and hash, key2 becomes the most significant
25 and key1 becomes the least significant part of the intermediate key material after concatenation.

The second input key may be a data encryption key or a KEK and must be untrusted.

The astute reader will note that three or more input keys may be combined by merging the output of one merge_long_key operation with yet another input key. One caveat to be observed is that when the concatenate operation is requested, the user must ensure that the sum of the two lengths of the input keys does not exceed the 64 byte maximum length of a long key.

Command Interface:

```

10  /* merge two keys and create a long key from the combined key material */
    cgx_merge_long_key( kernelblock    *kb,
                        longblob      *key1, /* either secretkey or longkey type */
                        crypto_cntxt  *kek1,
                        secretkey     *key2,
15  crypto_cntxt      *kek2,
                        UINT16        operation,
                        longkey       *lk,
                        crypto_cntxt  *bk_kek)

```

Arguments:

```

20  kb->cb->cmd      = CGX_MERGE_LONG_KEY;
    kb->cb->argument[0] = (VPTR)key1;
    /* the first input key to be merged (long key type or encryption key */

25  kb->cb->argument[1] = (VPTR)kek1;
    /* crypto context used to uncover key1 */

    kb->cb->argument[2] = (VPTR)key2;
    /* the second input key to be merged */

30  kb->cb->argument[3] = (VPTR)kek2;
    /* crypto context used to uncover key2 */

    kb->cb->argument[4] = (VPTR)operation;
35  /* CGX_CONCAT_O, CGX_XOR_O, CGX_HASH_O | CGX_SHS_A,
    CGX_HASH_O | CGX_MD5_A, or CGX_HASH_O | <other_hash_alg_type> */

    kb->cb->argument[8] = (VPTR)lk;
    /* pointer to result long key to be returned */

40  kb->cb->argument[9] = (VPTR)bk_kek;
    /* crypto context used to cover bk */

```

Status:

kb->status =

5

CGX_SUCCESS_S,
CGX_BAD_KEY_S,
CGX_NULL_PTR_S,
CGX_FAIL_S

EXTRACT(secret key from) LONG KEY

10

Command Name: CGX_EXTRACT_LONG_KEY

Command Description:

The CGX_EXTRACT_LONG_KEY command creates a secret key from key material supplied within a longkey.

15

The input longkey is presented to the function in the black; a crypto context for it must be supplied to uncover it internally. The resulting output secret key will be returned in the black so a crypto context for it must also be supplied as a function argument.

20

The material used to create the secret key consists of **length** bytes taken from the input long key starting at position **offset**. (Length and offset are parameters of the function call.) Naturally, the operation will fail if the input long key does not contain length+offset bytes.

The type and use of the result key are also specified as parameters.

Command Interface:

25

/* extract key material from long key and create secret key from it */

30

```
cgx_extract_long( kernelblock  *kb,
                  longkey      *key1,
                  crypto_cntxt *kek1,
                  UINT16       type,
                  UINT16       length,
                  UINT16       use,
```

```

        UINT16    offset,
        secretkey  *bk,
        crypto_ctxt *bk_kek)

```

5

Arguments:

```

        kb->cb->cmd      = CGX_LONG_KEY_EXTRACT;
        kb->cb->argument[0] = (VPTR)key1;
        /* the longkey container from which key info. will be extracted */

```

10

```

        kb->cb->argument[1] = (VPTR)kek1;
        /* crypto context used to uncover key1 */

```

15

```

        kb->cb->argument[2] = (VPTR)type;
        /* key type of result key to be produced */
        /* CGX_DES_A or CGX_TRIPLE_DES_A or other key type */

```

```

        kb->cb->argument[3] = (VPTR)length;
        /* desired length in bytes of result key to be created */

```

20

```

        kb->cb->argument[4] = (VPTR)use;
        /* attributes of key being created */

```

```

        kb->cb->argument[5] = (VPTR)offset;
        /* byte offset within key1 of first byte to extract */

```

25

```

        kb->cb->argument[6] = (VPTR)bk;
        /* pointer to result long key to be returned */

```

30

```

        kb->cb->argument[7] = (VPTR)bk_kek;
        /* crypto context used to cover bk */

```

Status:

```

        kb->status =

```

35

```

        CGX_SUCCESS_S,
        CGX_BAD_KEY_S,
        CGX_NULL_PTR_S,
        CGX_FAIL_S

```

CGX PRF DATA

40

Command Name: CGX_PRF_DATA**Command Description:**

The intended use of this command is to add data to the open inner hash context in an IPSec HMAC generation.

The CGX_PRF_DATA command hashes one, two or three data items, of different types, into the inner hash of an HMAC being generated: the items (in the order they are processed) are

- 5 - a secret key (specified by argument secretkey *bk)
- a g^{xy} DH shared key specified in argument publickey *gxypk (previously produced in the BLACK by cgx_negkey_gxy() for example.)
- 10 - RED data (specified in argument (VPTR)*dptr of a specified number of bytes (bytecount.)

If bk, gxypk, and/or dptr are NULL, the corresponding data are not processed.

15 If the secret key bk is not NULL, argument crpto_cntxt *bkek must specify the kek to uncover it. Also bk must be an untrusted key; it can be a data encryption key (K) or a KEK, etc.

 If gxypk is non NULL, argument gxykek must specify the key to uncover the g^{xy} key (presumably the same context used to cover gxypk in the first place.) Only the private key member (which holds the shared DH key) of gxypk and the modulus length of the public key are used in this operation. The other members of the public key need not be populated.

20

 Argument hash_cntxt *ihci specifies the previously initialized and still open source (input) hash context into which the data items will be hashed. (ihci MUST be supplied.) ihci can be RED or BLACK; if argument crypto_cntxt *ikek is non NULL, ikek will be used to uncover ihci. if ikek is NULL ihci is assumed to be RED.

25

Argument hash_cntxt *ihco specifies the destination context into which the result will be placed. (ihco may equal ihci, if desired) ihco can be returned RED or BLACK; if argument crypto_cntxt *okek is non NULL, okek will be used to cover ihco. if okek is NULL ihco will be returned in the RED.

okek, if specified, must be an untrusted key.

The resulting output hash context is not closed by this function. Repeated calls can be made to this function to add more data to the hash context by supplying the output hash context of one command invocation as the input hash context of a subsequent invocation.

Several variants of this command are provided. The first, cgx_prf_secretkey_gxy_data(), is the generic command described above in this section. The other variants merely provide convenient special case commands.

Command Interface:

```
/* hash additional data into open hash context */
cgx_prf_secretkey_gxy_data(
    kernelblock    *kb,
    hash_cntxt     *ihci,
    crypto_cntxt   *ikek,
    hash_cntxt     *ihco,
    crypto_cntxt   *okek,
    secretkey      *bk,
    crypto_cntxt   *bkek,
    publickey      *gxyPk,
    crypto_cntxt   *gxykek,
    void           *dpPtr,
    UINT16         bytecount)
```

Arguments:

```
kb->cb->cmd      = CGX_PRF_DATA;
kb->cb->argument[0].ptr = (VPTR)(ihci);
kb->cb->argument[1].ptr = (VPTR)(ikek); /* NULL allowed */
kb->cb->argument[2].ptr = (VPTR)(ihco);
```

```

kb->cb->argument[3].ptr = (VPTR)(okek);
kb->cb->argument[4].ptr = (VPTR)(bk);
kb->cb->argument[5].ptr = (VPTR)(bkek);
5 kb->cb->argument[6].ptr = (VPTR)(gxyprk);
kb->cb->argument[7].ptr = (VPTR)(gxykek);
kb->cb->argument[8].ptr = (VPTR)(dptr);
kb->cb->argument[9].ptr = (VPTR)(bytecount);

```

Status:

```

10 kb->status =
CGX_SUCCESS_S,
CGX_BAD_KEK_S,
CGX_NULL_PTR_S,
CGX_FAIL_S

```

15

```

/* macros that invoke particular, less general, cases of cgx_prf_secretkey_gxy_data */

```

20

```

/* hash a secret key into an open hash context */
#define cgx_prf_secretkey(kb, ihci, ikek, ihco, okek, bk, bkek)
cgx_prf_secretkey_gxy_data(kb, ihci, ikek, ihco, okek, bk, bkek, NULL, NULL, NULL,
NULL)

```

25

```

/* hash a shared DH private key into an open hash context */
#define cgx_prf_gxy(kb, ihci, ikek, ihco, okek, gxyprk, gxykek)
cgx_prf_secretkey_gxy_data(kb, ihci, ikek, ihco, okek, NULL, NULL, gxyprk, gxykek, NULL, NULL)

```

30

```

/* hash bytecount-many bytes of uncovered data into an open hash context */
/* If final != 0, this operation closes the hash context before returning it. */
#define cgx_prf_data(kb, ihci, ikek, ihco, okek, dptr, bytecount, final)
cgx_prf_secretkey_gxy_data(kb, ihci, ikek, ihco, okek, NULL, final, NULL, NULL, dptr, bytecount)

```

35

```

/* hash a secret key and a shared DH private key into an open hash context */
#define cgx_prf_secretkey_gxy(kb, ihci, ikek, ihco, okek, bk, bkek, gxyprk, gxykek)
cgx_prf_secretkey_gxy_data(kb, ihci, ikek, ihco, okek, bk, bkek, gxyprk, gxykek, NULL, NULL)

```

CGX PRF KEY

40

Command Name: CGX_PRF_KEY

Command Description:

Command CGX_PRF_KEY can be used to complete the IPSec HMAC. Command arguments supply two open hash contexts known as the inner hash context and the outer hash context, both of which are covered.

(Additional arguments supply the crypto contexts needed to uncover the hash contexts.) The command closes the inner hash context (its internal copy of the inner hash context – the caller's copy is not affected.) Then it hashes the digest of the inner hash context into the outer hash context. Then it closes the outer hash context (its copy of the outer hash context) and creates a secretkey of type specified by the caller from the outer hash digest and returns the key, covered, to the caller. It also leaves the created key in a specified key cache register, ready to use for encryption.

A second variant of this command does not create a secret key from the outer hash digest but rather returns the outer hash digest, still open, in the red, to the caller.

Command Interface:

/* Complete the IPSec HMAC – create a secret key from resulting hash digest */

```
cgx_prf_key(
    kernelblock    *kb,
    hash_cntxt     *ihci,
    crypto_cntxt   *ikek,
    hash_cntxt     *ohcntxt_in,
    crypto_cntxt   *okek,
    destkcr,
    UINT16         type,
    UINT16         length,
    UINT16         use,
    secretkey      *bk,
    crypto_cntxt   *bkek)
```

Arguments:

```
kb->cb->cmd      = CGX_PRF_KEY;
kb->cb->argument[0].ptr = (VPTR)(ihci);
kb->cb->argument[1].ptr = (VPTR)(ikek);
kb->cb->argument[2].ptr = (VPTR)(ohcntxt_in);
kb->cb->argument[3].ptr = (VPTR)(okek);
kb->cb->argument[4].ptr = (VPTR)(destkcr);
kb->cb->argument[5].ptr = (VPTR)(type);
kb->cb->argument[6].ptr = (VPTR)(len);
```

```

kb->cb->argument[7].ptr = (VPTR)(use);
kb->cb->argument[8].ptr = (VPTR)(bk);
kb->cb->argument[9].ptr = (VPTR)(bkkek);

```

Status:

```
kb->status =
```

```

CGX_SUCCESS_S,
CGX_BAD_KEK_S,
CGX_NULL_PTR_S,
CGX_FAIL_S

```

/* Second Variant: Complete the IPSec HMAC – return resulting outer hash context, still open, in the red. Do not create a secret key from resulting hash digest */

```
cgx_prf_hash(
```

```

    kernelblock    *kb,
    hash_cntxt     *ihci,
    crypto_cntxt   *ikek,
    hash_cntxt     *ohcntxt_in,
    crypto_cntxt   *okek,
    crypto_cntxt   *ohco_red)

```

Arguments:

```

kb->cb->cmd          = CGX_PRF_KEY;
kb->cb->argument[0].ptr = (VPTR)(ihci);
kb->cb->argument[1].ptr = (VPTR)(ikek);
kb->cb->argument[2].ptr = (VPTR)(ohcntxt_in);
kb->cb->argument[3].ptr = (VPTR)(okek);
kb->cb->argument[4].ptr = (VPTR)(NULL);
kb->cb->argument[5].ptr = (VPTR)(NULL);
kb->cb->argument[6].ptr = (VPTR)(NULL);
kb->cb->argument[7].ptr = (VPTR)(NULL);
kb->cb->argument[8].ptr = (VPTR)(NULL);
kb->cb->argument[9].ptr = (VPTR)(ohco_red);

```

Status:

```
kb->status =
```

```

CGX_SUCCESS_S,
CGX_BAD_KEK_S,
CGX_NULL_PTR_S,
CGX_FAIL_S

```

ACRONYMS/TERMS

5

ACRONYMS

The following table is a list of acronyms used with the description of the co-processor.

Acronym	Meaning
ASIC	<i>Application-Specific Integrated Circuit</i>
CBC	<i>Cipher Block Chaining (a block cipher mode)</i>
CFB	<i>Cipher FeedBack (a block cipher mode)</i>
CGX	<i>CryptoGraphic eXtensions (IRE's crypto library)</i>
CT	<i>Cipher Text</i>
DEK	<i>Data Encryption Key</i>
DES	<i>Data Encryption Standard</i>
D-H	<i>Diffie-Hellman</i>
DMA	<i>Direct Memory Access</i>
DSA	<i>Digital Signature Algorithm</i>
ECB	<i>Electronic Code Book (a block cipher mode)</i>
FIFO	<i>First-In, First-Out</i>
HMAC	<i>Hash Message Authentication Code</i>
IV	<i>Initialization Vector</i>
KCR	<i>Key Cache Register (an internal key storage location)</i>
KEK	<i>Key Encryption Key</i>
LSV	<i>Local Storage Variable</i>
MAC	<i>Message Authentication Code</i>
OFB	<i>Output FeedBack (a block cipher mode)</i>
PCDB	<i>Program Control Data Bits</i>
PIN	<i>Personal Identification Number</i>
PT	<i>Plain Text</i>
SHA	<i>Secure Hash Algorithm</i>
SHS	<i>Secure Hash Standard</i>

10

TERMS

The following table is a list of terms used with the description of the co-processor.

Term	Meaning
Black Key	<i>A secret/private key that is encrypted or covered by a KEK, it can be securely given to another party.</i>
Covered Key	<i>A secret key that has been encrypted, via a KEK, to protect the key from being seen by an untrusted party. Same as a Black key or a Wrapped key.</i>
Export	<i>In the CryptIC, exporting a key refers to the process of covering the key with a KEK and then providing it to an external application in the 'Interoperable External Form'.</i>
Import	<i>In the CryptIC, importing a key refers to the process of uncovering the key from its 'Interoperable External Form' and storing it in an internal Key Cache Register.</i>
Key Cache Register (KCR)	<i>A working storage area for secret keys, addressable via a register ID between 0 and 14.</i>
Key RAM (KRAM)	<i>A volatile public key work area. The public key will be lost during a power-down or reset.</i>
Local Storage Variable (LSV)	<i>A non-volatile Laser-programmed secret key that can be used by the application as its own unique private key. Each CryptIC has a unique LSV programmed into it at the factory.</i>
Off-Load	<i>In the CryptIC, off-loading a key refers to the process of covering the key with a KEK and then providing it to an external application in the 'IRE External Form'.</i>
Program Control Data Bits (PCDB)	<i>Programmable control bits to customize the CGX Kernel features (such as allowing Red key exportation/importation, LSV changes, exportable chip, etc.).</i>
Public Keyset	<i>Specifies the related key pair (Public and Private) which make up a 'public key'</i>
Red Key	<i>A secret/private key that is not encrypted or covered by another KEK. It is in its raw unprotected form.</i>
Salt	<i>Random data which precedes a message in order to randomize its</i>
Symmetric Key	<i>A key which is used in a ciphering algorithm where the encrypting key and the decrypting key are the same (eg. DES)</i>
Wrapped Key	<i>A secret key that has been encrypted, via a KEK, to protect the key from being seen by an untrusted party. Same as a Black key or a Covered key.</i>

5 Although illustrative embodiments of the present invention have been described herein with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected by one skilled in the art without departing from the scope or spirit of the invention.

**Section I. Extending Cryptographic Services to the Kernel Space of a Computer
Operating System**

Background of the Invention

Field Of The Invention

5

This invention relates to computer operating systems for a personal computer or the like, and more particularly relates to computer operating systems which provide cryptographic services.

Description Of The Prior Art

10

There is currently on the market software for personal computers which provide cryptographic services. In particular, Microsoft Corporation provides its CryptoAPI (TM) software for its Windows (TM) operating system. The cryptoAPI (TM) software is a modular way to provide cryptographic (e.g., encryption) services to applications. For example, an E-mail encryption package on one's personal computer running in Windows (TM) will most likely be using the services CryptoAPI (TM) to perform the encryption processes.

15

20

CryptoAPI (TM) software is designed to be modular in that it includes a generic layer plus a replaceable library of encryption algorithms, referred to as a cryptographic service provider (CSP) module. The CSP module is software which is implemented in the form of a dynamic linked library (DLL) residing in the application space of the operating system. The CSP module contains many encryption algorithms, such as DES, triple DES, hashing algorithms, digital signature algorithms, etc. Since algorithms may change, and the rules of cryptography may change, the CSP module may be replaced with an updated version having new encryption

algorithms. The new CSP module is designed to be compatible with the generic layer of the CryptoAPI (TM) program.

CryptoAPI (TM) software operates only in the application space of the operating system of the personal computer (PC). Therefore, it can only be called upon by an application, such as E-mail, MicroSoft Word (TM), Excel (TM), or the like.

The CryptoAPI (TM) software cannot work in the kernel space of the operating system. The kernel space is that layer of the operating system which is essentially non-visible to the user, in other words, at the driver level of the PC, for example, where IP (Internet Protocol) packets are processed, where the disc drive controller software resides, where the PC's printer drivers are located, etc.

Kernel space routines cannot cross the line into application space very efficiently and use the services of CryptoAPI (TM) software in the application space. Therefore, if one wants to encrypt data or instructions coming in or out of the hard drive, the CryptoAPI (TM) software would not be usable, as it resides in the application space and not in the kernel space. Similarly, the IP packets would also not be able to be encrypted using the CryptoAPI (TM) software, as the IP packets are processed in the kernel space.

Objects And Summary of The Invention

It is an object of the present invention to define an implementation of cryptographic services in the kernel space of a computer operating system.

It is another object of the present invention to define the implementation of cryptographic services in the kernel space of a computer operating system which is linked to similar cryptographic services provided in the application space.

It is still another object of the present invention to provide an implementation of cryptographic services for an operating system usable in a personal computer which is capable of encrypting hard drive data and IP packets at the driver level of the personal computer.

5 In accordance with one form of the present invention, a cryptographic service software is embodied in at least one of a hard disc, a floppy disc or a read-only memory (ROM). The cryptographic service software electronically communicates and is compatible with a standard operating system of a computer, such as MicroSoft Windows (TM). The operating system includes an application space and a kernel
10 space. The cryptographic service software performs cryptographic services at the kernel space of the operating system. The cryptographic service software includes a generic layer having a kernel space level program interface, and a cryptographic service module having a library of encryption algorithms. This module may be replaced with a different module having updated or at least different encryption
15 algorithms.

In another form of the present invention, cryptographic service software is situated in each of the application space and kernel space of a standard operating system for a computer. The separate application space and kernel space software are linked together to exchange cryptographic functions, such as algorithms, digital
20 signatures and hash functions and secretkey material. Each of the application space and kernel space cryptographic software includes a generic layer having a program interface, and a cryptographic service module having a library of encryption algorithms, which module electronically communicates with the program interface. Each module is preferably replaceable, as mentioned previously.

25 These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawings.

Brief Description of The Drawings

Figure 36 is a block diagram illustrating the implementation of a cryptographic service software in accordance with the present invention.

Detailed Description of The Preferred Embodiments

5 In accordance with one form of the present invention, cryptographic service software similar in operation and structure to the CryptoAPI (TM) software sold by MicroSoft Corporation is preferably embodied in either a hard disc 3, floppy disc 2 or a read-only memory (ROM) 4. The ROM 4 or hard disc 3 may be situated in a personal computer 6 or other piece of electronic equipment, and the floppy disc 2 may
10 be received and read by a disc drive of the computer 6 or other equipment.

 The cryptographic service software is compatible and communicates with a standard operating system of a computer, such as the Windows (TM) operating system. Unlike the CryptoAPI (TM) software, the cryptographic service software of the present invention is situated in the kernel space of the operating system, at the
15 driver level of the computer. The cryptographic service software performs cryptographic services using encryption algorithms and the like at the kernel space of the operating system.

 The cryptographic service software is structured similarly to that of the CryptoAPI (TM) software. It includes a generic layer having a kernel space level
20 program interface 8, which functions and operates in a manner similar to the application program interface of the CryptoAPI (TM) software. It further includes a cryptographic service module 10 which may be embodied in a similar manner to that of the CryptoAPI (TM) software. The cryptographic service module 10 preferably includes a library of encryption algorithms. The module electronically communicates
25 and cooperates with the kernel application programming interface 8. This module

may be replaced with a different module having new or different encryption algorithms.

The cryptographic service software allows one to write code at the driver level of the computer in a manner similar to the way the CryptoAPI (TM) software does at the higher, application level. Now, encryption algorithms may be used to encrypt signals at the driver level, such as at the Ethernet port or at the modem port, video card or disk drive, etc., that is, at a level where the conventional CryptoAPI (TM) software cannot reach. The cryptographic service software at the driver level is still accessible by application software 12 through secured drivers (engines) 14 situated at the driver level. Also, advantageously, during software development the cryptographic software code at the kernel level may be debugged at the application level.

Preferably, and as shown in the figure, a cryptographic service software is situated at each of the application space and the kernel space, and the two are linked together. Each cryptographic service software may be loaded from a floppy disc 2 onto a computer 6 or may be embodied in a read only memory (ROM) 4. The application space software includes an application program interface 16 and a cryptographic service module 18 electronically communicating with the application programming interface 16. The kernel space cryptographic software includes a kernel space level program interface 8 and a cryptographic service module 10 electronically communicating with the application programming interface 8. Each of the cryptographic service modules 10, 18 preferably includes a library of encryption algorithms and the like. Preferably, the modules are linked together to exchange algorithms, for example, or share secret key material between the two. This link facilitates the operation of the computer 6 and the exchange of encrypted material from one computer to another because the application level cryptographic software may wish to use the same pre-arranged keys in its application level communications as are used at the kernel level such as for encrypting and decrypting IP packets.

In another form of the present invention, it is envisioned that there are a plurality of security enabled kernel engines 14 situated in the kernel space. These security enabled kernel engines 14 communicate with and drive various components, such as a disk drive 20, hard drive 22 and internet port 24 of the computer. Each security enabled kernel engine 14 electronically communicates with preferably the same kernel space program interface 8 of the kernel cryptographic service software. The advantage in this arrangement is that the cryptographic service software and module thereof may be shared by many different kernel engines 14 as opposed incorporating in each kernel engine an encryption algorithm. Of course, there may be unsecured engines 26 situated in the kernel space communicating with and driving other components 28 for which cryptographic services are not required. Nevertheless, each of the security enabled kernel engines 14 and unsecured engines 26 communicate with the application software 12.

Preferably, the kernel space cryptographic service software and, in particular, the kernel space program interface 8 thereof, electronically communicates with other hardware crypto devices, such as the cryptographic co-processor 30 disclosed in the patent application entitled "Cryptographic Co-Processor" filed concurrently herewith, the disclosure of which is incorporated herein by reference. The cryptographic co-processor 30 has mask-programmed in a memory 32 thereof a library of encryption algorithms and the like. Accordingly, the cryptographic service software situated at the kernel space is linked not only to the cryptographic service software situated at the application level, but also to a hardware cryptographic device, such as the co-processor 30 mentioned previously. Therefore, the application software may utilize the cryptographic library in the kernel space, which is preferably pure software, or the cryptographic library in the co-processor 30, which is essentially hardware.

Although illustrative embodiments of the present invention have been described herein with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

Cryptographic service software embodied on a hard disc or a floppy disc electronically communicates with a standard operating system of a personal computer. The operating system has an application space and a kernel space. The cryptographic service software performs cryptographic services at the kernel space of the operating system. The cryptographic service software includes a kernel space level application programming interface and a cryptographic service module having a library of encryption algorithms.

Section II. Method of Communicating Securely Between an Application Program and a Secure Kernel

Background of the Invention

5 **Field Of The Invention**

The present invention relates generally to a method of communicating securely between an application program and a secure kernel, and more particularly relates to a method of passing command requests and arguments between an application program and a secure kernel so that security intensive and real time intensive
10 applications can coexist without a security breach.

Description Of The Prior Art

Passing software structures such as data and commands between subroutines within a program is known in the art. Software is typically created in a modular form where each module communicates with each other by passing information back and forth. Software pointers and hardware registers to locate data within a program are
15 also commonly used. Calling subroutines and jumping from one software module to another module are also known in the art. These software techniques are applied to all types of application programs including secure communication programs. However, these application programs do not have distinct boundaries between the software
20 modules and the security software modules. An application program be permitted to enter a secure program area. There are software techniques to implement this process in a secure manner. However, it can never be truly secure because once an application program command structure or data structure is permitted to enter a secure program module, the application program module is free to do whatever a
25 programmer instructs it to do, such as retrieve secret cryptographic keys.

Objects and Summary of the Invention

It is an object of the present invention to provide a method of communicating securely between an application program and a secure kernel.

5 A method of securely communicating between an application program and a secure kernel that contains cryptographic algorithms includes the step of creating a memory storage area for storing and reading command requests, command data and status data. The address of the memory storage area is passed to the secure kernel. The secure kernel proceeds to bring in only the information within the memory storage area that it deems is necessary and the application program is not permitted to enter the secure kernel. The secure kernel processes the information retrieved, using
10 cryptographic algorithms located within the secure kernel and transfers results back to the kernel block where the application program can retrieve them.

Since all data transfers across the boundary between the application program and the secure kernel are initiated by the kernel, a high degree of protection is
15 afforded against attacks such as: trojan horse, illicit commands, virus, etc. The kernel is always fully in control of the commands and data it processes.

These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawings.

Brief Description of the Drawings

20 Figure 37 is a block diagram of portions of an integrated circuit having secure and unprotected memory areas used for securely communicating between an application program and a secure kernel in the integrated circuit in accordance with the present invention.

Figure 38 is a flowchart of the method of passing command requests and arguments between an application program and a secure kernel in accordance with the present invention.

Detailed Description of the Preferred Embodiments

5 In accordance with one form of the present invention, and as shown in Figure 1, an integrated circuit, preferably a cryptographic co-processor such as that disclosed in U.S. Patent Application entitled "Cryptographic Co-Processor", the disclosure of which is incorporated herein by reference, includes a kernel block memory structure 2, a command block memory structure 4 and a register 6 which together provide a
10 boundary between an application program 8 and a secure kernel 10, which includes cryptographic algorithms and commands. All communications between the application program 8 and the secure kernel 10 are through the kernel block memory 2 and the command block memory structure 4. The kernel block 2 and the command block 4 are used by the application program and the secure kernel to transfer
15 command communications between the application program and the secure kernel. The kernel block 2 and command block 4 store command codes, arguments, and status and pointer information. Together, the kernel and command blocks occupy a pre-formatted shared block of random access memory (RAM) located in either an internal data memory or an external data memory.

20 The application program 8 stores the required cryptographic commands and arguments in the command block memory 4. They are stored in this separate area of memory because the application program is not permitted to directly enter the secure kernel 10. The commands are tools which allow the application software to control and manipulate the cryptographic algorithms located within the secure kernel 10. The
25 arguments are the data which is passed between the application program 8 and the cryptographic algorithms. The command block memory structure 4 is used as a data transfer area between the application program 8 and the secure kernel 10 for servicing cryptographic commands.

Preferably, the kernel block 2 includes a status block field 12. The status block field 12 provides the application program 8 with the means to track the status of the secure kernel 10 (e.g. active or idle) and a means to determine the result of a requested cryptographic service. As shown in the flowchart of Figure 2, the application program first sets up the kernel block 2 and command block 4, preferably allocating 21 16 bit words of RAM for the command block 4 and 4 16 bit words of RAM for the kernel block 2 (Block 2).

The register 6 stores the starting address location of the kernel block 2 so that the secure kernel 10 can locate the kernel block memory 2 at a later time. The command block memory 4 and the kernel block memory 2 can be located anywhere within the processor's data memory. However, the starting address of the kernel block memory 2 should be stored in the register 6.

The command block memory 4 and kernel block memory 2 may be populated with the desired command and data (Block 4). The starting address of the kernel block 2 is stored in the register 6. To execute a cryptographic service, the application program 8 must explicitly invoke the secure kernel 10. Invoking the secure kernel 10 is achieved by executing a secure kernel transfer vector (e.g., call 0x2000 instruction).

Using the call (i.e., command) instruction (e.g., call 0x2000), the application program 8 relinquishes processor control to the secure kernel's read only memory (ROM) encoded program (Block 6). Address 0x2000 is preferably the only entry point of the secure kernel 10. The starting address of the kernel block memory 2 is located in the register 6 which is passed into the secure kernel 10 (Block 8). The secure kernel 10 uses the address to locate and then retrieve the kernel block memory information. The kernel block memory information contains a pointer to a command address where the command block is stored. The secure kernel 10 proceeds to save the caller's context (application program context) in a memory stack (Block 10). This preferably involves saving the caller's program counter (PC) return address, stack pointer, and frame pointer. Once the context has been saved, the secure kernel 10

points to the data memory address where the kernel block memory 2 is stored and proceeds by retrieving the commands and arguments stored within the kernel block 2 and command block 4 (Block 12). The secure kernel 10 processes the commands and data using the cryptographic algorithms and commands (Block 14). It then transfers
5 the processed information back to specified locations in data memory and updates the status field in the kernel block 2, at which point the application program 8 can now access the processed information.

Although illustrative embodiments of the present invention have been described with reference to the accompanying drawings, it is to be understood that the
10 invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A method of communicating securely between an application program and a
15 secure kernel is performed by passing command requests and arguments between the application program and the secure kernel through a kernel block memory and a command block memory so that security intensive and real time intensive applications can co-exist without a security breach. The secure kernel retrieves the command requests and the arguments from an application program data memory and processes
20 the information within the secure kernel. The secure kernel returns the processed data to the application program. All data transfers are under control of the secure kernel software, and thus numerous 'active attacks' against the security of the system are defeated.

Background of the Invention**Field Of The Invention**

The present invention relates generally to a secure memory area, and more particularly relates to a secure area of memory with multiple communication buses having hardware that prevents unauthorized access to each communication bus.

Description Of The Prior Art

Application programs and data stored within a memory circuit are typically protected by an operating system software, if protected at all. The software allocates memory to an application program and prevents the application program from executing instructions outside the allocated memory space. Preventing application programs from exiting the designated memory space indirectly creates a secure environment within the memory circuit.

Software memory protection is not entirely secure because there is no hardware to physically block access to a particular area of memory. With software memory protection, it is possible have private data or encryption algorithms sharing a memory device with public information. Even though software protection isolates memory space between two application programs, it remains physically possible to access the private information.

Objects and Summary of the Invention

It is an object of the present invention to provide a secure memory area for storage of cryptographic keys, algorithms and data having security hardware that prevents unauthorized access to each storage area.

A secure memory area constructed in accordance with one form of the present invention includes a main communication bus circuit and one or more separate secondary memory bus circuits. The main communication bus circuit and secondary bus circuits, and any related memory circuits, are preferably formed on a single monolithic integrated circuit (chip). The secondary memory bus circuits preferably include a key bus circuit. The key bus circuit is provided for isolating a secret key storage area from the external world (i.e., anything outside the chip, for example, commands from an unauthorized accessor). This eliminates the possibility of accidentally leaking secret key material to the outside world. Another preferred secondary bus circuit is a cryptographic algorithm bus circuit. The cryptographic algorithm bus circuit is provided to eliminate the risk of an outside source from accessing cryptographic algorithms stored in a memory circuit coupled to the cryptographic algorithm bus circuit such as via an external memory bus circuit. A third preferred secondary bus circuit is the external memory bus circuit which has coupled to it one or more external memories (for storage of application programs, for example). Bus transceivers are coupled between each individual secondary communication bus and the main communication bus. Security is established by providing separate secondary communication buses for public and private information.

Brief Description of the Drawings

Figure 39 is a block diagram of a secure cryptographic memory area formed in accordance with the present invention.

Detailed Description of the Preferred Embodiments

A block diagram of the secure cryptographic memory area formed in accordance with the present invention is illustrated in Figure 1. The secure memory area preferably has three sections: key memory 2, external memory 4, and internal memory 6.

A first bus transceiver 8 is coupled to a key bus circuit 30. The first bus transceiver 8 controls access between the key bus circuit 30 and a main bus circuit 42. The key bus circuit 30 is coupled to a key random access memory (RAM) 12, a key cache memory 10, and a factory laser bit storage memory 14. The factory laser bit storage memory 14 stores a unique factory set variable used to encrypt keys. The first bus transceiver 8 is coupled between the main bus circuit 42 and the key bus circuit 30. This isolates the key bus circuit 30, and all memories and sections connected thereto, from the main bus circuit 42. A separate bus circuit ensures that when encryption services are operating on memory circuits coupled to the key bus circuit 30, data (e.g. a secret key) cannot be leaked to the external memory 4. This is prevented by having the external memory 4 on a separate external memory bus circuit 32. Access to the external memory bus circuit 32 is controlled by a second bus transceiver 18, which cannot be activated at the same time that the first bus transceiver 8 is activated.

The key RAM 12 provides a public key volatile storage area. The key RAM 12 has enough space to accommodate the private portion of at least one active public key operation. The key RAM 12 can not be read by an external application because,

while the external memory 4 is being accessed, the first bus transceiver 8 blocks access to the key RAM 12.

The key cache memory 10 allows the application to access preferably up to 15 volatile secret key cache memory locations in which are stored various encryption keys. Each key cache location is preferably 30 words in length. The external application can not directly read the key cache memory 10 because of the bus isolation provided by the first bus transceiver 8.

The external memory bus circuit 32 couples an external RAM 20 and an external read only memory (ROM) 22 to the main bus 42 through the second bus transceiver 18. The second bus transceiver 18 controls access to the external memory bus circuit 32 from the main bus circuit 42. Having a separate external memory bus circuit 32 is important because, while the outside world is accessing the main bus circuit 42, the first bus transceiver 8 prevents access to the key bus circuit 30 and the secure key data stored in memory.

A third bus transceiver 24 controls access between the main bus circuit 42 and a cryptographic algorithm bus circuit 40. The cryptographic algorithm bus circuit 40 couples a scratch RAM 26 and an internal ROM 28 to the third bus transceiver 24. A separate bus is provided to prevent secure data and algorithms from being accessed by an external source via the external bus circuit 32. An external application can not read the internal ROM 28 because the third bus transceiver 24 is deactivated when the second bus transceiver 18 is activated. The third bus transceiver 24 is also deactivated when the first bus transceiver 8 is activated.

External RAM 20 is used to store application software for use by a processor. Encryption algorithms are stored in the internal ROM 28. Commands are passed back and forth between ROM 28 (encryption kernel) and the application via the external RAM 20. When the processor is accessing the external memory bus circuit 32, it is not possible to access the internal ROM 28 because it is isolated by the third bus

transceiver 24. This prevents an external device, such as an emulator, from accessing the internal ROM 28 and reading the secure algorithms.

A small scratch RAM 26 exists for the encryption kernel and the cryptographic services to use as a storage device for intermediate calculations. The scratch RAM 26 is isolated from the external applications by the third bus transceiver 24.

The main communication bus 42 is coupled to a digital signal processor (DSP) 16, which internally includes a microprocessor. The microprocessor of the DSP 16 preferably communicates with and controls the activation and deactivation of the bus transceivers 8, 18, 24 by sending control signals to each transceiver. The DSP 16 ensures that only one transceiver will be active at any given time.

Hardware protection eliminates the possibility of compromising private algorithms or data. Isolating memory circuits and external devices with separate communication buses increases security and lowers the risk of accidentally releasing private information. Structuring memory around separate communication buses and permitting only one communication bus to be accessed at a time provides hardware security that exceeds that provided by software.

Although illustrative embodiments of the present invention have been described with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A hardware secure memory area includes one or more secondary communication buses connected to a main communication bus. The secondary communication buses are coupled to the main communication bus by separate bus

transceivers. The bus transceivers provide isolation between the communication buses and between unaccessed secondary buses and the main communication buses. Various external devices, such as memories, may be coupled to the communication buses. Only one bus transceiver may be activated at a time, thus making it impossible for two secondary communication buses to be linked.

Section IV. Method of Expanding Protected Memory in an Integrated Circuit

Copyright Notice

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyrights whatsoever.

Background of the Invention

Field Of The Invention

The present invention relates generally to a method of expanding protected memory, and more particularly relates to a method of expanding memory protection into an unprotected memory area while providing protection to the newly acquired memory area.

Description Of The Prior Art

Typical operating systems have memory protection systems. They provide an application program with a designated memory area and prevent the application program from executing instructions outside the area. When an application program is launched, the operating system allocates sufficient memory for the application to

run. The operating system prevents the application program from going outside the allocated memory area. If the application tries to execute an instruction outside the allocated memory area, the operating system shuts down the application. A secure memory environment is created by preventing each application program from exiting its own designated memory area. This prevents applications from entering other application memory space, thus creating a secure environment. This approach is good if one wants to prevent applications within the system from entering other application memory space. However, it also prevents an application program from executing instructions outside of its allocated memory area and does not provide any flexibility to the amount of memory allocated for the application program.

Objects and Summary of the Invention

It is an object of the present invention to provide a method of expanding a protected memory area into an unprotected memory area, while providing protection to the newly acquired memory.

It is another object of the present invention to provide a method of expanding protected memory in increments and allowing an application to execute instructions outside the protected memory area.

It is another object of the present invention to provide a method of permitting an application program to enter and exit a protected memory area.

A method of expanding memory protection into an unprotected memory area in accordance with one form of the present invention includes the step of requesting initialization of a secure kernel (which includes an operating program in a secure memory area) by an application program. The step of specifying a starting block address and number of blocks to be protected provides maximum flexibility to the application program using the integrated circuit having protected (secure) and unprotected memories. The memory is partitioned into blocks, thus providing the

OEM with the flexibility to expand the protected memory in increments. The starting block address and the number of blocks are stored in a hardware register. The ability to store this information in register provides the application with the ability to jump in and out of the new protected memory.

5 The application can relinquish the newly acquired protected memory by requesting the secure kernel to reinitialize itself. This involves clearing the memory that contains the starting block address and the number of blocks so that on power up the secure kernel is in a default setting. It also results in the expanded protected memory being overwritten with a random pattern. This destroys the protected
10 information that was previously stored in the memory.

These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawing.

Brief Description of the Drawing

15 Figure 40 is a flowchart of the method of expanding protected memory into unprotected memory.

Figure 41 is a block diagram illustrating various memories used in an integrated circuit formed in accordance with the present invention.

Detailed Description of the Preferred Embodiments

20 A flowchart of the method of expanding protected memory into an unprotected memory formed in accordance with the present invention is illustrated in Figure 1. The method provides a mechanism for expanding a secure key cache memory space

into an unprotected memory space, while extending security protection to the new memory space.

A secure kernel program is mask-programmed into preferably a 32K word (word = 16 bits) read only memory (ROM) located within a general purpose processor of an integrated circuit (IC). A volatile random access memory (RAM) is also preferably provided for the storage of encryption keys. The secure kernel allots memory space in the volatile key RAM for at least one public key and a plurality (preferably 15) of secret keys. To accommodate additional encryption keys and key data, an OEM in whose end product (e.g. modem, router, cellular phone, etc.) the IC is used may expand the size of the key cache memory by expanding into the internal processor data memory space, which would have otherwise been used by an application program. The data memory is partitioned into preferably 1K word (word = 16 bits) blocks comprising the address range which can be protected.

As shown in the flowchart of Figure 40, the request to expand the secure key cache memory space into the unprotected data memory space requires that the application program include an application program interface (API) call (e.g. CGX INIT command) to initialize the secure kernel (Block 2). The initializing step is similar in many respects to when one "boots up" a computer. A self-test is preferably performed by the chip, various housekeeping chores are attended to, stack pointers are reset, certain memories are erased and memory that was previously locked in as being allocated as protected memory is released and the contents erased. This command causes the arguments attached to the command to be vectored into the secure kernel where the secure kernel reads the arguments. The arguments include pointer data and the number of new protected memory blocks requested. The pointer data contains the starting memory address of the requested new protected memory (i.e. expanded key cache). The secure kernel receives the command and reads the arguments (Block 4). The secure kernel then goes to the starting memory address and locks in the number of 1K word memory blocks requested (Block 6). The memory is locked in, for example, by writing a "1" into a data memory reserve register located in the

protected kernel space (Block 8). For every 1K word block of memory requested, a "1" is preferably written into the data memory register. The data memory reserve register is preferably 16 bits long, as there are preferably 16 1K word blocks of memory which may be designated as protected.

5 Figure 41 shows the various memories and reserve registers used in an integrated circuit (IC) formed in accordance with the present invention, and illustrates the technique for expanding the protected memory in the IC. The invention has particular application to a cryptographic co-processor, such as described in U.S. Patent Application entitled "Cryptographic Co-Processor" filed concurrently herewith,
10 the disclosure of which is incorporated herein by reference.

 Figure 41 illustrates the data memory 2 and its associated reserve register, referred to herein as the DM reserve register 4. As can be seen from Figure 2, and as described previously, when a 1K word (word = 16 bits) block of memory is reserved for being protected, a "1" is written into the associated data memory register at a bit
15 place in the register corresponding to the particular block (page) of memory reserved.

 The program memory 6 and the key memory 8 of the integrated circuit, and more particularly, the cryptographic co-processor mentioned above, may also be used for expansion of the secure kernel in a manner similar to that described with respect to the data memory 2. In other words, the program memory 6 has associated therewith a
20 reserve register, referred to as a PM reserve register 10, and the key memory 8 has associated therewith a reserve register, referred to as the KM reserve register 12.

 Each of the PM reserve register 10 and the KM reserve register 12 is also preferably 16 bits long, and each bit in the register corresponds to a particular page or block of memory in the program memory 6 and key memory 8, respectively.

25 With respect to the program memory, there are 16 1K word (word = 24 bits) blocks of memory which may be reserved as protected. With respect to the key

memory 8, there are 16 256-byte blocks of memory which may be reserved as protected. Preferably, the memory locking registers are only available (read/write) when in the kernel (protected) mode.

5 Memory blocks designated as protected by the application remain protected until memory or chip power is removed, a chip or system reset occurs, or a command to restore the protected memory to a default setting (i.e., memory allocation prior to the expansion) is generated.

10 A computer program showing the method of re-configuring an integrated circuit in accordance with the present invention is provided herewith and is incorporated herein as part of the disclosure of the invention.

15 Although illustrative embodiments of the present invention have been described herein with reference to the accompanying flowchart, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

20 A method of expanding memory protection into an unprotected memory area includes the step of setting an address within the unprotected memory where the new protection will be provided. The original equipment manufacturer (OEM) selects the starting memory block address and the number of memory blocks required for the expanded memory location. The information is stored in a memory and read by a secure kernel during initialization. After initialization the secure kernel adjusts its memory boundaries according to the request of the OEM.

Section IV. Method for Expanding Secure Kernel Program Memory

Copyright Notice

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

Background of the Invention

Field Of The Invention

The present invention relates generally to a method of expanding a secure kernel memory area, and more particularly relates to a method of expanding a secure kernel memory area into an unprotected memory area while testing for validation and providing protection to the newly acquired memory area.

Description Of The Prior Art

Software developers attach a digital signature to their software code to protect users from code that has been modified. The modification may occur during or after the manufacturing process. Digital signatures are attached to each software package during the final stages of the manufacturing process. Each signature has a data item which accompanies a digitally encoded message and is used to determine if the code has been modified. Before the user is permitted to load the entire software package on to a computer, the digital signature must be checked for authenticity. This is accomplished by comparing the digital signature within the code to a digital signature provided by the user. If the software code has been tampered with or a computer virus

has attacked the code, the digital signature within the code will be altered. A difference between the two digital signatures indicates that data integrity has been breached and the software is prevented from being loaded into the computer.

Objects and Summary of the Invention

It is an object of the present invention to provide a method of expanding a secure kernel memory area into an unprotected memory location while testing for validation and providing protection to the newly acquired memory area.

It is an object of the present invention to provide a method for adding new authorized encryption algorithms to a secure kernel while providing the new algorithms with the same security as mask-programmed cryptographic algorithms.

It is another object of the present invention to provide flexible memory protection that can only be accessed by a super user, for example, the manufacturer of the integrated circuit having the protected and unprotected memories.

It is an object of the present invention to provide a manufacturer with flexibility and control over the addition of code to an existing system.

A method of expanding a secure kernel memory area formed in accordance with the present invention includes the step of signing an application program or encryption algorithm with a digital signature. This is required so that the manufacturer of an integrated circuit (IC) containing a secure kernel memory can control code that is added to the secure kernel memory. It also prevents unauthorized access to the secure memory area. The IC manufacturer generates a digital signature using its private key. The digital signature is verified by the secure kernel in the end product (e.g., router, modem, cellular phone) in which the IC is being used using a public key, which is stored in a read only memory (ROM) within the IC. The secure

kernel verifies the digital signature and if it is valid, the secure kernel locks the expanded memory into protected mode and loads the new code. If the signature is invalid, the request is denied.

Brief Description of the Drawing

Figure 42 is a flowchart of a method of expanding a secure kernel program memory space.

Detailed Description of the Invention

A flowchart of the method of expanding a secure kernel memory area into an unprotected memory area formed in accordance with the present invention is illustrated in Figure 42. The method provides a mechanism for expanding a secure kernel memory area into an unprotected memory area, while providing security to the newly acquired memory and validation of new kernel code.

A secure kernel program is mask programmed into preferably a 32K word (word = 24 bits) read only memory (ROM) located within a general purpose processor. The processor has preferably 16K words (word = 16 bits) of data random access memory (RAM) and 16K word (word = 24 bits) of program RAM available for applications. The kernel ROM is protected by hardware that prevents direct access to the kernel by an application. The hardware provides a shared memory area where commands and arguments can be passed between the application and the secure kernel. To accommodate an additional cryptographic algorithm or other kernel extension, the IC manufacturer or an original equipment manufacturer (OEM) of whose product the IC is a component (such as in a router, modem, cellular phone, etc.) may expand the size of the secure kernel memory space. This is accomplished by expanding into the internal processor program memory space and/or data memory space, which would otherwise have been used for an application program. The data

memory and the program memory are preferably partitioned into 1K word (word = 16 bits) blocks.

An additional cryptographic algorithm or other kernel extension (i.e., extended code) is required to be signed by a trusted authority before it can be down loaded into the newly acquired memory. This method employs a digital signature to authenticate the kernel extension by embedding the signature in a token which is presented along with the new code. Each integrated circuit having the secure kernel memory and unsecured memory is identified by a unique factory programmed identification code, making it possible to single out a specific integrated circuit. The creation of the kernel extensions is controlled to prevent the down loading of illegal or untrusted cryptographic algorithms into the integrated circuit (IC).

Referring to Figure 42, to down load the new cryptographic algorithm, or other kernel extension, the application software first requests the serial number of the IC using a command provided within the IC, as shown in Block 2. The preferred serial number is a hash of a local storage variable (LSV). The LSV is a unique variable that is set by burning fuses within the IC during the manufacturing process. Once the OEM retrieves the serial number, the OEM sends the serial number and the extended code to the manufacturer of the IC, as indicated in Block 4. In Blocks 6 and 8, if the manufacturer approves of the extended code, a token is generated and sent to the OEM. The token includes the serial number, digital signature of the extended code created by the IC manufacturer. Having the serial number of the IC in the returned token is preferred so that only the IC requested to be extended in secure memory will be permitted by the IC manufacturer to do so, and not a different IC. If the IC manufacturer does not include the serial number in the return token, this preferably means that it approves a secure memory expansion for all ICs. In Block 10, the OEM application program loads the extended code and the token into an internal program random access memory (RAM) located within the IC.

The application program, in Block 10, executes a command (e.g., CGX_LOAD_EXTENDED) which transfers the extended code into the secure kernel. All interrupts are disabled to maintain the integrity of the process and to prevent someone from accessing the secure kernel and obtaining secure code. The secure kernel takes control of the process and fetches the token and the serial number (Block 12). A pointer to where the extended code is located in memory is passed into the secure kernel. The amount of memory requested and the token are also passed into the secure kernel with this command. In Block 14, the token is separated and the serial number field is extracted from the token. If a serial number is present, it is compared to the serial number within the IC, as shown in Blocks 16 and 18. If the serial numbers do not match, then the token was not directed to this IC and the process is aborted. This feature allows the IC manufacturer to target the extended code to selected ICs. Preferably, as shown in Block 16, if there is no serial number found when the token is parsed, it means that the extended code is approved to be downloaded into any or all ICs made by the IC manufacturer. If the serial numbers match, or there is no serial number, then the signature verification phase may begin.

The signature verification phase is a standard process which may be achieved in several different ways. In the preferred method, the kernel hashes the extended code together with the serial number and computes a Digital Signature Algorithm (DSA) digital signature of the hash, as indicated in Blocks 20 and 22. In Block 24, the computed signature block is verified against the signature passed in with the token. If they do not match, the process is aborted, as shown in Block 26. If they do match, then the code is considered to be valid and the secure kernel locks in the approved number of program memory pages, as indicated in Block 28. This is preferably achieved by setting a bit in the program memory reserve register to 1, for every 1 K word block of program memory space used for the extended code. In Block 30, the secure kernel marks the extended code as valid and the code is ready to execute.

A computer program illustrating one form of the method of the present invention is provided herewith and is incorporated herein as part of the disclosure of the invention.

Although illustrative embodiments of the present invention have been described herein with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A method of expanding a secure kernel memory area to accommodate additional software code includes the step of digitally signing the additional code by a trusted authority. The code has a digital signature to authenticate the source of the code and to control what code can be added to the secure kernel. The new code is copied into an unprotected memory where the digital signature is verified. The digital signature includes a unique integrated circuit (IC) identification number, which provides the IC manufacturer with the ability to control the secure kernel memory expansion of all or each of the ICs. If the code is authenticated via the digital signature, then those memory blocks are locked-in as protected memory and thus given "secure kernel" privileges.

Section VI. Method of Reconfiguring the Functionality of an Integrated Circuit

Copyright Notice

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent

and Trademark Office patent files or records, but otherwise reserves all copyrights whatsoever.

Background of the Invention

Field Of The Invention

The present invention relates generally to a method of enabling features of an integrated circuit, and more particularly relates to a method of controlling configuration settings and program upgrades to an integrated circuit contained in a product released to the field.

Description Of The Prior Art

Products that are developed having multiple features or configurations are typically set manually by a manufacturer or a product user. This usually requires setting an internal dual in-line package (DIP) switch. This approach may be convenient and require limited support hardware; however, it does not provide the manufacturer with control over what features or configurations the user selects. Another technique used by manufacturers to vary their product features is to provide different programmable read only memory (ROM) circuits for every product variation. This approach prevents users from changing the product configuration, but it also eliminates the flexibility of changing the configuration of the product in the field.

Objects and Summary of the Invention

It is an object of the present invention to provide a method of changing configuration settings and program upgrades in an integrated circuit from a remote location only by an authorized person.

A method of enabling features within an integrated circuit (IC) in an authenticated manner formed in accordance with the present invention includes the step of retrieving a unique serial number stored in memory within the IC. The original equipment manufacturer (OEM) in whose end product (e.g., modem, router, cellular phone, etc.) the IC is used requests a token (i.e., authorization signal) from an IC manufacturer to upgrade features within the IC. If the IC manufacturer decides to permit the OEM to upgrade the features of the IC, the token will be sent to the OEM with new configuration bits, the serial number and a digital signature to authenticate the source of the data. The returned serial number is verified in the IC to confirm that the IC receiving the upgrade approval is the IC which made the request. The digital signature is also verified to confirm that it is the manufacturer of the IC who is approving the upgrade. If the serial number is verified and the digital signature is verified, then a kernel (i.e. operating system) stores the new configuration bits in a memory, and the IC is reconfigured as requested by the OEM and authorized by the IC manufacturer.

These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawing.

Brief Description of the Drawing

Figure 43 is a flowchart of the method of the present invention for controlling and setting different features within an integrated circuit.

Detailed Description of the Preferred Embodiments

The features of the invention described herein are suitable for use in many different integrated circuits, but are particularly applicable for use in a cryptographic co-processor, such as that disclosed in the U.S. patent application entitled "Cryptographic Co-Processor" filed concurrently herewith, the disclosure of which is incorporated herein by reference. A number of terms used herein are defined in the aforementioned patent application, and reference should be made to such application for a more detailed explanation of such terms.

An integrated circuit (IC) is manufactured with preferably 256 programmable fuses which are programmed by a laser during the manufacturing process. Of the 256 fuses, 48 make up program control data bits (PCDBs) which enable and disable various IC features and configure the IC. The configuration in which the fuses are programmed determine what security features are activated within the IC (e.g., key lengths, red KEK loading, public key vector size and algorithm enable). Preferably, these security features are already present in the chip, and only some may have been activated. None of the PCDB features have to be enabled during the manufacturing process; however, in this case the original equipment manufacturer (OEM) would have to request a feature-enable token from the IC manufacturer to enable the cryptographic functions. There is a unique IC serial number embedded within every IC that is used to identify the IC.

The functionality and features of the IC can only be modified with approval from the IC manufacturer. This is to prevent unauthorized use of encryption

algorithms in the preferred use of this method in a cryptographic co-processor. Each integrated circuit is designed and manufactured with numerous encryption features (e.g., DES, 3DES, etc.) However, not every encryption feature may be activated when the IC is shipped to the OEM. Activation may depend upon the OEM's requirements or the OEM's geographic location. A particular integrated circuit may be manufactured to provide only a limited level of security to meet United States Government export laws. If the laws change or the product incorporating the IC is moved inside the United States, the IC may be enabled in the field to operate at a greater level of security.

Referring to Figure 43, to request an encryption upgrade or enable new features of the IC, the OEM application software executes a command that retrieves the serial number from the IC (e.g., CGX_GET_CHIPINFO) (Block 2). Once the serial number is retrieved, it can be communicated to the IC manufacturer, along with the PCDB settings needed to reconfigure the IC, as a token request signal (Block 4). If the IC manufacturer approves of the OEM's request to upgrade the cryptographic functions (Block 6), then the IC manufacturer will create a return token signal to enable the requested upgrades (Block 8). The token is created by the IC manufacturer and contains the new PCDBs. A digital signature of the IC manufacturer is attached to the return token and communicated to the OEM with the unique IC serial number (Blocks 8 and 10). The serial number is included in the return token so that only the designated IC can be upgraded. The digital signature is generated by the IC manufacturer using a private key which is only known to the IC manufacturer. The OEM application program receives the return token which includes the digital signature and serial number, and submits them to a secure kernel in the IC (Block 12 and 14). The secure kernel includes cryptographic algorithms and digital signature algorithms. The kernel parses the token to separate the serial number and the digital signature, and compares the parsed serial number to the serial number embedded within the chip (Blocks 16 and 18). If they do not match, the process is aborted. If they do match, the kernel enters a signature verification phase to verify that the parsed digital signature is from the IC manufacturer (Block 20). This process preferably uses

a standard SHA-DSA algorithm using a public key stored within an IC non-volatile memory. If the digital signature does not verify correctly, the process is aborted. If the signatures match, the kernel stores the new PCDBs in a memory (Block 22). On power up or IC reset, the kernel will read the new PCDBs and disregard the old, burned-in PCDBs and re-configure the IC to the appropriate settings. With the example of a cryptographic co-processor, the OEM manufacturer may now use the IC with the authorized encryption algorithms or hash functions.

A computer program showing the method of re-configuring an integrated circuit in accordance with the present invention is provided herewith and is incorporated herein as part of the disclosure of the invention.

Although illustrative embodiments of the present invention have been described herein with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A method of reconfiguring the functionality of an integrated circuit (IC) includes the step of retrieving a serial number embedded in the IC and transmitting a token request signal to an authorizing party. The token request signal includes the serial number and a reconfiguration code to be used for reconfiguring the IC. The authorizing party transmits a return token signal to the IC which includes the serial number, the reconfiguration code and a digital signature of the authorizing party. The return token signal is parsed by the IC to extract the serial number, reconfiguration code and digital signature of the authorizing party. The parsed serial number and signature are verified by the IC as being correct, and the reconfiguration code is stored in a memory of the IC. The IC then reconfigures itself in accordance with the reconfiguration code.

Section VII. Method of Implementing a Key Recovery System

Copyright Notice

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

Background of the Invention

Field of the Invention

The present invention relates generally to a method of encryption key recovery on an integrated circuit, and more particularly relates to a method of establishing a trusted key relationship with an authorized party which allows a user to recover an encryption key in a secure manner.

Description of the Prior Art

Key recovery is typically used to retrieve a copy of a private key when the key is lost, or is unknown to an employer, or when a court order has granted a government agency the right to monitor communication traffic. A lost key results in lost data because without the key, the encrypted data cannot be decrypted. A disk holding the key may be lost or a hardware failure may result in a lost key. A user needs to have the ability to recover a lost key. An employer may need to recover private keys generated by former or disgruntled employees to retrieve corporate information. A government authority may need a private key to observe an encrypted data transmission when there is a suspicion of criminal activity. The typical method of key recovery includes sending a wrapped copy of the private key with each transmission. Then under the appropriate circumstances, this key may be unwrapped with a recovery key.

Objects and Summary of the Invention

It is an object of the present invention to provide a method of creating a recovery key encryption key (RKEK) in a secure manner so that only an authorized party can own the RKEK.

5 It is another object of the present invention to create an RKEK for wrapping keys used in an encryption process, and for recovering the encryption key and decrypting data at a later date.

 In accordance with one form of the present invention, a method of generating a recovery key encryption key (RKEK) in a secure manner by an integrated circuit (IC) and a key recovery escrow agent includes the steps of generating by the IC a first
10 number having a private component and a public component, and generating by the escrow agent a second number having a private component and a public component. The method further includes the steps of providing the public component of the first number to the escrow agent, and providing the public component of the second
15 number to the IC.

 Then, a Diffie-Hellman modulo-exponentiation mathematical operation is performed by the IC using the private component of the first number, and the public component of the second number to create the RKEK. Also, the escrow agent performs a similar operation, that is, a Diffie-Hellman modulo-exponentiation
20 mathematical operation using the private component of the second number, and the public component of the first number to create the RKEK at its end.

 These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawing.

Brief Description of the Drawing

Figure 44 is a flowchart of a method in accordance with the present invention for creating a secure recovery key encryption key.

Detailed Description of The Preferred Embodiments

5 The following technique describes an approach for encryption key recovery that meets the needs of both the government and users. It provides a way for an application program to establish a trusted key relationship (a secure protocol) with a third party, such as an integrated circuit (IC) manufacturer or a certifying/escrow agency, so that a recovery key encryption key (RKEK) can be created. This provides
10 the escrow agency with the means to get at any key created/protected by the IC.

 In accordance with one form of the present invention, a random number is burned into a read only memory (ROM) on the integrated circuit (IC) by the manufacturer of the chip. This random number is a unique serial number which is used to identify the chip.

15 With respect to key recovery, the purpose is to have the recovery key encryption key (RKEK) embedded in the IC and used as part of the normal encryption routines performed by the chip. The RKEK is used to wrap or encrypt other keys used in the encryption process. One always wants to ultimately protect the keys used in encryption, and would never want to let keys be exported outside the chip, except
20 keys will be allowed to leave the chip if they are protected by the RKEK.

 The RKEK will be embedded in the chip, but also the idea is to have someone else have a copy of the RKEK, i.e., the escrow agent or the "key recovery agent". This will allow someone else (other than the chip, or more precisely, the OEM manufacturer in whose end product (e.g., router, modem, cellular phone, etc) the chip
25 is found) to decrypt the data or recover the key used in the encryption process.

First, the RKEK must be generated and it is preferably stored on the chip in a key cache register. The RKEK is a key that is in the IC which is used to “wrap” other keys used in the encryption process. Once one has created an RKEK in the chip, one wants to create a carbon copy of it for the escrow agent to hold.

5 The OEM product manufacturer, who manufactures a device, such as a router or modem in which the encryption chip is used, usually has an agreement with an escrow agent. The escrow agent and the OEM manufacturer agree on the modulus and generator used by the chip. The modulus and generator are public elements (i.e., numbers) used in public key cryptography. If two parties want to take part in a public
10 key operation, including creating the RKEK, then the parties must agree on the modulus and generator so that the two parties will be, in effect, communicating in the same language. Once the OEM product manufacturer and the escrow agent have decided on the modulus and generator used in the chip, the application software uses a command, such as CGX_GEN_NEWPUBKEY, to begin the process of generating a
15 public key.

 In generating a public key, the chip preferably uses a Diffie-Hellman (D-H) public key process, although one can use RSA, elliptic curve and other well-known public key algorithm techniques. The following explanation of the RKEK process will be described using the Diffie-Hellman (D-H) public key method. The D-H public
20 key method is preferred over elliptic curve and RSA for generating the RKEK because each party contributes equally to the generation of the RKEK and no one party has an advantage over the other.

 With the D-H public key method, each party to the communication will end up having a key (i.e., a relatively long number) which will be the same. Each party starts
25 out with its own number that it chooses. The number has a private component and a public component. Each party exchanges (reveals to the other) its public component. In the case of generating the RKEK, one party is the OEM product manufacturer in whose product the encryption chip is used, and the other party is the escrow agent.

After the exchange, each party ends up having its own private part, its own public part and the other party's public part.

In accordance with the D-H method, a mathematical operation (modulo-exponentiation arithmetic) is performed by each party, using an exponential formula ($g^{xy} \bmod n$). By using this mathematical operation on the private components and the known public components, each party can derive the same number (key). An outsider (eavesdropper) to the communication has only access to the two public parts and neither private part and, therefore, is denied access to the key.

With the command, CGX_GEN_NEWPUBKEY, the IC will create a number having a private part and a public part. The escrow agent does the same. The private part stays protected on the chip; it never leaves the chip. The escrow agent's number also has a private part and a public part (the escrow agent carefully holds in confidence the private part). The numbers are generated by both parties using the D-H modulus and generator.

Now, the IC generates a request token (i.e., message) to generate an RKEK. In the token is preferably repeated the unique serial number of the chip and the public part of the D-H key set (and optionally a hash of this data for integrity purposes). This request token is preferably sent to the chip manufacturer (i.e., a trusted third party), which acts as a middleman between the OEM part manufacturer (whose product uses the IC) and the key escrow agent.

Thus, the manufacturer of the IC has the public component of the IC's key, and the serial number. The manufacturer recognizes the chip from the serial number and may verify with the escrow agent that a key recovery process has been agreed to between the OEM product manufacturer and the agent. The IC manufacturer then authorizes the creation of the RKEK.

Preferably, the chip will not be able to create the RKEK without the chip manufacturer's authorization. Using the chip manufacturer as the middleman to give approval to create the RKEK adds an extra measure of security to further preclude an unauthorized adversary from creating an RKEK that may be used to decrypt data and uncover the encryption key.

The chip manufacturer "signs" the request token by adding its digital signature using a private key. The signed token, which preferably comprises the serial number (of the chip), the public component (referred to as " $g^x \text{ mod } n$ ") of the ICs recently generated public keyset and the IC manufacturer's digital signature (which hashes all of the other data in the token) is forwarded to the IC as well as to the escrow agent.

The application software of the IC uses a new command, for example, CGX_GEN_RKEK, to pass into the chip the signed token as an argument to the command. An additional argument is the public key component from the escrow agent.

The public key component from the escrow agent may be delivered directly from the escrow agent to the IC or may be routed through the trusted third party.

The chip checks the token's digital signature using a public key burned into the IC during manufacturing to verify the signature of the IC manufacturer, and further checks the serial number in the returned signed token to see if it matches that which has been burned into the chip during manufacturing. If both portions check out, then the token has been validated.

The RKEK is created by the chip from the escrow agent's public key component (which the chip now has) and the chip's private key component (which it has been holding onto). The same RKEK is also created by the escrow agent from its private key component, and the chip's public key component which it received from the chip either directly or through the chip manufacturer.

The ultimate RKEK which is created is the D-H shared secret, i.e., a modulo-exponentiation operation is performed using the other party's public key and the first party's private key (x or y). The result of this operation is a number that both parties will have, but which an eavesdropper cannot generate. This number becomes the RKEK.

As a result of this exchange of information, the chip now has the RKEK, which is stored in its key cache register, the escrow agent has the same RKEK which it saves, and the chip manufacturer or trusted third party has no RKEK because it was not privy to the private key components of the two other parties.

Alternatively, the escrow agent may choose to not generate the RKEK immediately, but rather to store the IC's public key so that it can generate the RKEK in the future should it be necessary.

The preferred integrated circuit uses many different keys, such as KEK's (key encryption keys), RKEK's, DEK's (data encryption keys), LSV's (local storage variables), and others. Each one of these keys has an attribute which identifies what type key it is and whether it is a trusted or untrusted key. The key management software of the IC reads these attributes and, therefore, recognizes the various keys, including the RKEK, and knows that it can use the RKEK as a key encryption key to encrypt other types of keys and allow them to be exported out of the chip.

The RKEK is used to "wrap" other keys which are used for data encryption, and the wrapped key may be exported with the encrypted data. Therefore, for exported encrypted data, if a receiver of the data cannot find the original encryption key, the key was exported with the data, and therefore, all the receiver needs is the RKEK. Accordingly, the RKEK may be used to encrypt data (by wrapping the encryption key) but also for decrypting data (by recovering the encryption key).

The preferred method of generating a recovery key encryption key (RKEK), in accordance with the present invention, is shown in the flow chart of Figure 44 and will now be described in detail. The integrated circuit (IC) is referred to in the flow chart by the trademark CryptIC, and the term "IRE" refers to the assignee and owner of the invention, Information Resource Engineering, Inc. IRE is the manufacturer of the integrated circuit and is the trusted third party in the operation of generating an RKEK.

In accordance with the preferred method, the first step in the process is to have the integrated circuit and the recovery agency generate a public key set. Preferably, as mentioned previously, a Diffie-Hellman (D-H) public key set is used. The steps involved in generating the D-H public key set are shown in the flow chart and labeled as Blocks 2-10.

First, the application software on the integrated circuit requests the serial number from the integrated circuit (Block 2). This is done with a command, such as CGX_GET_CHIPINFO. The next step is for the application software associated with the integrated circuit to transmit the serial number to the recovery agency (Block 4). This is done with a message, which is referred to as a "request" or "token".

Both the integrated circuit (perhaps under control of, the OEM manufacturer) and the recovery agency agree on a particular modulus "m" and generator "g", and the recovery agency returns its modulus and generator to the application software of the integrated circuit (Block 6). The recovery agency also generates a new D-H public key set (Block 8). Similarly, the integrated circuit uses the modulus "m" and the generator "g" to generate a D-H public key set (Block 10). This is usually done through a command by the application software, such as CGX_GEN_NEWPUBKEY.

The application software for the integrated circuit then constructs a key-recovery request token message and sends this message to the trusted third party (Block 12). The request token preferably includes the integrated circuit serial number,

which is the unique number which is programmed into the integrated circuit, and the D-H public key of the integrated circuit. The trusted third party (for example, IRE, the manufacturer of the integrated circuit) signs the request token with its private signature key (i.e., a digital signature) and returns the token as a message to the integrated circuit (Block 16). This return token preferably includes the integrated circuit serial number, the D-H public key (of the integrated circuit) and the trusted third party's digital signature.

The recovery agency sends its public key to the integrated circuit application software (Block 14). Now, the integrated circuit has all that it needs to generate the RKEK.

The application software commands the integrated circuit to generate the RKEK (Block 18). It may do this with a command, such as CGX_GEN_RKEK. The CGX kernel (secure portion of the integrated circuit) parses the recovery token sent from the trusted third party (IRE) so that the trusted third party's digital signature and the serial number may be verified (Block 20). The integrated circuit then compares the serial number in the recovery token from the third party with its own serial number programmed in the chip to see if there is a match (Block 22). If the two serial numbers do not match, then the routine is aborted and the request to generate an RKEK is rejected (Block 24). If the serial numbers match, then the integrated circuit verifies whether the digital signature from the trusted third party is authentic by using a trusted public key which the integrated circuit has stored in memory (Block 26). If the digital signature is not authentic, then the routine is aborted and the request to generate an RKEK is rejected (Block 28). If the digital signatures match, then the integrated circuit will calculate an RKEK using the D-H algorithm (Block 30).

Similarly, the recovery agency calculates its copy of the same RKEK (Block 32). The recovery agency was sent the public key of the integrated circuit (Block 16) and, from this public key, and its private and public key, the recovery agency uses the Diffie-Hellman algorithm to generate the same RKEK at its end. The recovery

agency then stores its RKEK along with the integrated circuit's serial number in escrow (Block 34).

A computer program showing the operation of the integrated circuit in generating the RKEK in accordance with the present invention is provided herewith and is incorporated herein as part of the disclosure of the invention.

Although illustrative embodiments of the present invention have been described herein with reference to the accompanying drawing, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A method of generating a recovery key encryption key (RKEK) in a secure manner by an integrated circuit (IC) and a key recovery escrow agent includes the steps of generating by the IC a first number having a private component and a public component, and generating by the escrow agent a second number having a private component and a public component. The public component of the first number is provided to the escrow agent, and the public component of the second number is provided to the integrated circuit. A Diffie-Hellman modulo-exponentiation mathematical operation is performed by the integrated circuit using the private component of the first number, the public component of the first number and the public component of the second number to create the RKEK. A similar operation is performed by the escrow agent using the private component of the second number, the public number of the second number and the public component of the first number to create the RKEK at its end.

Section IIX. Kernel Mode Protection

Background of the Invention

5 The present invention relates generally to kernel mode protection, and more particularly relates to an apparatus and method that enforces a security perimeter around cryptographic functions.

Description of the Prior Art

10 The concept of privileged separation of software processes is known in the art. Having one software process work in a privileged/secure environment and another software process working in an unprivileged/non-secure environment is typically controlled by the hardware within a processor. The hardware that supports privileged separation is fundamentally integrated throughout the processor mask. Integrating the hardware throughout the processor is expensive and increases the size of the processor. Smaller processors, such as digital signal processors (DSPs), do not
15 include privileged separation hardware features. The hardware is not included in the processors to keep the size and cost of the processors down.

Brief Description of The Drawings

Figure 45 is a drawing of the kernel mode protection circuit.

Figure 46 is a flow chart of a method of kernel mode protection.

Objects And Summary of The Invention

20 It is an object of the present invention to provide an apparatus and method that enforces a security perimeter around cryptographic functions.

The kernel mode protection circuit constructed in accordance with one form of the present invention includes a processor, a program counter, a kernel fetch
25 supervisor circuit, a kernel data fetch supervisor circuit, a program memory, a data

memory, a flip-flop circuit and two AND circuits. The kernel mode protection circuit may operate either in a user mode or a kernel mode. The kernel program fetch supervisor circuit monitors the address within the program counter and compares the address to a predetermined address stored within the kernel program fetch supervisor. If the addresses are equal the kernel program supervisor circuit activates a flip-flop which switches between a user mode output signal and a kernel mode output signal. The kernel data fetch supervisor circuit compares the processor data address to a predetermined protected memory address range. If the processor is in user mode and attempts to fetch data within the protected memory address range, then the kernel data fetch supervisor circuit in conjunction with the flip-flop circuit generates a processor reset signal at the AND circuit output. If the processor attempts to access a kernel memory address other than the predetermined address stored in the kernel program fetch supervisor circuit, then the kernel program fetch supervisor circuit in conjunction with the flip-flop will generate a processor reset signal at the AND circuit output.

Detailed Description of the Preferred Embodiments

The kernel mode protection circuit, Figure 45, is responsible for enforcing a hardware security perimeter around cryptographic functions. The circuit may either be operating in user mode (kernel space is not accessible) or kernel mode (kernel space is accessible) at a given time. When in the kernel mode the kernel random access memory (RAM) and certain protected registers and functions (kernel space) are accessible only to the secure kernel firmware. The kernel executes host requested macro level functions and then returns control to the calling application. The kernel mode control hardware subsystem will reset the processor should any security violation occur, such as attempting to access a protected memory location while in user mode. Any attempt by a user mode application program running on the processor to access a kernel space address other than 0x2000 will result in an immediate processor reset and all sensitive registers and memory locations will be erased. Kernel mode may only be entered via a call, jump or increment to address

0x2000. However, while in kernel mode, the processor 4 may access all program/data memory and registers.

The kernel mode protection circuit, Figure 45, includes the following: a processor 4, a program counter circuit 6, a kernel program fetch supervisor circuit 8, a flip-flop circuit 10, an AND circuit 12, a program memory 14, a kernel data fetch supervisor circuit 20, a data memory 22 and an AND circuit 30. The program memory 14 includes a user memory 16, a kernel read only memory (ROM) 18 and a user memory 20. The data memory 22 includes a user memory 28, protected registers and random access memory (RAM) 24 and a user memory 26.

The program counter (PC) 6 is coupled to the kernel program fetch supervisor circuit 8 and the program memory 14. The program counter 6 contains the address of the current program fetch instruction. The kernel program fetch supervisor circuit 8 contains a basic comparator used to determine whether the PC 6 is set to address 0x2000 or another address range. The kernel program fetch supervisor circuit 8 is coupled to the flip-flop circuit 10 by an access user output 50 connected to the flip-flop circuit 10 set input, and an access kernel 0x2000 output 60 is coupled to the flip-flop circuit 10 clear input. The flip-flop circuit 10 has 2 outputs, a user mode output 52 and a kernel mode output 54. The kernel program fetch supervisor circuit 8 also has an access kernel not = 0x2000 output 62. This output and the user mode output 52 are coupled to a standard AND circuit 30. The kernel program fetch supervisor circuit 8 operates in three states. The first state occurs when the processor 4 is in the user mode and a program fetch is from a user program memory 14. The second state occurs when the processor 4 is in the user mode and enters the kernel at address 0x2000. The third state occurs when the processor 4 is in the user mode and an application program tries to access the kernel using ROM 18, an address other than 0x2000.

The kernel data fetch supervisor circuit 20 is coupled to a data memory address bus 64 and the data memory 22. An access kernel data output signal 66

couples the kernel data fetch supervisor circuit 20 to the AND circuit 12. The kernel data fetch supervisor circuit 20 compares the data memory address fetch to the address range of the protected registers and RAM 24. The address range is preferably 0000 through 17FF. If the data address fetched is within the address range 0000 through 17FF, the kernel data fetch supervisor circuit 20 asserts a logic "1" signal. In addition, if the processor 4 is in user mode (a logic "1" signal at user mode output 52), a logic "1" signal is generated from the AND circuit 12. This signal resets processor 4 since fetching data from protected memory, while in user mode, is not permitted.

The first state occurs when the processor 4 is in user mode and a program fetch is from a user program memory. The kernel program fetch supervisor circuit 8 compares the program counter 6 address to address 0x2000. If the addresses are equal, then the access user output 50 sets flip-flop 10 to kernel mode. If the addresses are not equal then the kernel program fetch supervisor circuit 8 does nothing and the processor stays in user mode.

The second state occurs when the processor 4 is in user mode and the processor 4 tries to access the kernel at address 0x2000. The kernel program fetch supervisor circuit 8, compares the address stored in the program counter 6 to address 0x2000. If they are equal, the kernel protection fetch supervisor circuit 8 activates the kernel output 60 and clears the flip-flop 10 resulting in the processor 4 switching to kernel mode.

The third state occurs when the processor 4 is in user mode and an application program tries to access the kernel at an address other than 0x2000. The kernel program fetch supervisor circuit 8 compares the address stored in the program counter 6 to the kernel address not equal 0x2000. If the address is within the kernel space but is not equal to 0x2000, then the output 62 is set to a logic "1" the flip-flop 10 to user mode.

The kernel data fetch supervisor circuit 20, compares a data address fetch to the address range of the protected registers and RAM 24. This region of memory is preferably from address 0000 through to address 17FF. If the data address is within this range, the kernel data fetch supervisor circuit 20 sets an output logic "1" signal on the access kernel data output 66.

User mode output 52 is coupled to the AND circuit 12 input and the AND circuit 30 input. The access kernel not = 2000 output 62 is coupled to the AND circuit 30. The access kernel data output 66 is also coupled to the AND circuit 12.

When the processor 4 is in user mode, the kernel program fetch supervisor circuit 8 is in the access user state. This state sets the user mode output 52 to a logic "1" signal. If the processor 4 attempts to access an address other than 0x2000, the kernel program fetch supervisor circuit 8 generates an output logic "1" signal on the access kernel not = 2000 output 62. Jumping from user mode to kernel mode, using an address other than 0x2000, is an illegal operation (violation of the security features). These signals, when applied to the AND circuit 30, result in a logic "1" signal at the AND circuit 30 output which resets the processor.

If the processor 4 is in the user mode and is then put into kernel mode at address 0x2000, the kernel program fetch supervisor circuit 8 clears the flip-flop 10 and applies a logic "1" signal to the access kernel output 60. The flip-flop 10 also applies a logic "0" signal to the user mode output 52, which disables the AND circuits 12 and 30. These circuits are disabled to prevent the reset signal from being generated, because accessing the kernel at address 0x2000 from user mode is permitted.

Access kernel data output 66 and user mode output 52 are coupled to the AND circuit 12 inputs. If the processor 4 is in the user mode, the user mode output 52 is a logic "1" signal. If the processor 4 tries to access data within the protected data memory range, then a logic "1" signal is generated on the access kernel data output 66 and the AND circuit 12 generates a reset signal which resets the processor 4. The

processor 4 is reset because fetching data from protected registers and RAM 24, while in user mode, is an illegal operation. However, while the processor 4 is in kernel mode, the data fetch is permitted to be from anywhere within the data memory 22. A logic "0" signal is generated on the user mode output 52. This disables the AND circuit 12, which prevents the processor from being reset.

A flowchart of a method of kernel mode protection is shown in Figure 46. The method starts with the processor operating in user mode (Block 2). An application program operating outside of the kernel is considered to be operating in the user mode. While in user mode, the application program fetches program opcodes (Block 4). The number of opcodes fetched depend upon the particular application program that is running. Each opcode fetch is checked whether it was fetched from kernel memory or application memory (Block 6). If the opcode fetch is from the kernel memory, this is a violation and the processor is reset. If the opcode fetch is from user memory, the process continues. Each data operand fetch is monitored by the secure kernel (Block 8). If the data operand fetch is from kernel memory then the processor 4 is reset (Block 10). If it is from user memory the process continues. When the processor code makes a call to address 0x2000 it enters the secure kernel and switches to kernel mode (Block 12 & 14). While in kernel mode, all program fetch opcodes are also monitored (Block 16). If the opcode fetch is from kernel memory, block 16, then the data operand may also be fetched from either kernel or user data memory (Block 18 & 20). This process continues until the application is complete or if an opcode fetch is from user memory (Block 16 & 18). If the opcode fetch is from user memory the processor switches back to user mode (Block 2).

Abstract of The Disclosure

A kernel mode protection circuit includes a processor, a program counter, a kernel program fetch supervisor circuit, a kernel data fetch supervisor circuit, a program memory, a data memory, a flip-flop circuit and two AND circuits. The data memory includes two user memories, protected registers and random access memory (RAM). The program memory includes two user memories and a kernel read only memory (ROM). The circuit may operate in either a user mode (kernel ROM is not accessible) or a kernel mode (kernel ROM is accessible). When in the kernel mode the kernel RAM and certain protected registers are accessible only by a secure kernel. The kernel mode control circuit will reset the processor should a security violation occur, such as attempting to access the kernel RAM while in the user mode. The kernel program fetch supervisor circuit monitors and compares an address within the program counter to a predetermined address, stored within the kernel program fetch supervisor circuit, to determine if a security violation has occurred. The kernel data fetch supervisor circuit monitors and compares the data address to addresses defining a protected memory area. A security violation will occur if the data address is within the protected memory address range and the processor will be reset. A method of kernel mode protection includes the step of fetching a program opcode. If the program opcode is from the kernel memory, the processor is reset. If the program opcode is from a user memory, then the processor may fetch the data operand. If the data operand is fetched from the kernel memory, the processor is reset. If the data operand is fetched from a user memory, the processor is permitted to enter the kernel memory. If a program opcodes is fetched from the kernel memory the processor may continue to fetch operands from either the kernel memory or the data memory. The processor remains in kernel mode and continues to fetch program opcodes until all of the opcodes have been fetched, or until an opcode fetched is from the user memory. If an opcode fetched is from the user memory, the processor switches back to user mode.

Section IX Cryptographic Key Management Scheme

Copyright Notice

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

Background of the Invention

Field Of The Invention

The present invention relates generally to a cryptographic key management scheme, and more particularly relates to a method of creating and manipulating encryption keys without risking the security of the key.

Description Of The Prior Art

All cryptographic techniques, whether used for encryption or digital signatures depend upon cryptographic keys. Cryptographic key management systems are a crucial aspect of providing security. Typical key management systems include key generation policies, key distribution policies and key termination policies. Cryptographic key management schemes are built on commonly used lower level concepts, such as key wrapping techniques. These techniques vary with the type of cryptographic algorithm used and are vast in numbers. However, key management systems are unique to the developer and vary substantially depending on the type of key used and level of security required.

The key management scheme allows for efficient access to all keys so that the cryptographic algorithms can run as fast as possible and be as compact as possible, with little secure tradeoffs as possible.

Object and Summary of the Invention

It is an object of the present invention to provide a comprehensive powerful and secure encryption key management scheme.

5 It is another object of the present invention to provide the user with a set of encryption key management rules and commands for various encryption key algorithms.

It is a further object of the present invention to provide a method of managing the use of keys in a cryptographic co-processor.

10

It is still another object of the present invention to provide the user with a set of encryption key management rules that prevent the user from generating risky keys.

15

In accordance with one form of the present invention, a method of managing the use of keys in a cryptographic co-processor includes the steps of selecting a key type from one of a symmetrical key type and an asymmetrical key type. Then, the key bit length is selected. The key is then generated and, lastly, the key is represented in either an external form or an internal form.

20

The key management method allows for many different key types to be selected. Also, several key lengths may be chosen. The key may be generated in various ways to meet industry standards, and the key may be represented preferably in either an inter-operable external form or internal or external forms set up by the cryptographic co-processor manufacturer.

25

The key management scheme allows for a wide range of key management implementations. It is only concerned with supplying primitive key management utilities. This allows the application using the encryption key management to create

either a simple flat key management structure, or a highly layered and complex military grade key management system.

Brief Description of the Drawings

Figure 47 is a flowchart of the method for creating and manipulating encryption keys.

Detailed Description of the Preferred Embodiments

Section I of the Detailed Description

The features of the invention described herein are particularly applicable for use in a cryptographic co-processor, such as that disclosed in the U.S. patent application entitled "Cryptographic Co-Processor" filed concurrently herewith, the disclosure of which is incorporated herein by reference. A number of terms used herein are defined in the aforementioned patent application, and reference should be made to such application for a more detailed explanation of such terms.

Referring to Figure 47, it will be seen that the key management method described in accordance with one form of the present invention includes the steps of: selecting a key type, selecting a key bit length, generating a key, and representing a key.

The key management method requires the user to first select between several different key types. The key management method supports algorithms and key sizes for symmetrical (secret keys) and asymmetrical (public key) cryptographic operations. The symmetrical encryption algorithms supported are DES, Triple DES (two key), and Triple DES (three key). The key sizes are 40-56 bits for DES, 40-112 bits for Triple DES (two key), and 40-192 bits for Triple DES (three key). It also supports asymmetric encryption such as Diffie-Hellman and RSA. The key size for each is preferably between about 512 and about 2048 bits. Digital signatures such as DSA and RSA are also supported and have key lengths of preferably between about 512 and about 2048 bits.

The first step (Block 2) requires the user to select either a symmetrical type of key or an asymmetric type of key. If the user selects a symmetrical type of key, there are four types of symmetrical keys for the user to choose from: data keys, key encryption keys, program keys, and message authentication code keys. The step of
5 key access is preferably automatically selected with the choice of key type.

Data Encryption Keys (DEKs) allow a message, communications channel and files to be encrypted using one of the block algorithms (i.e. DES, Triple DES, etc.) supported by the user's system. All DEKs are stored and retrieved from volatile key cache registers. By default, a DEK is not permitted to be exported (i.e. read by the
10 application) unless it is covered by a key encryption key. However, the DEK must be uncovered in the user's system to be used as a traffic key while it is active.

A key encryption key (KEK) allows keys to be exported for storage (as in key escrow) to allow key distribution, or for the secure exchange of traffic keys. The key management scheme preferably supports three types of KEKs: an internally generated storage variable (GKEK); a local storage variable (LSV); and a user application
15 generated KEK (KEK).

The application KEK is a KEK that is not preserved across resets and its storage requirements must be implemented by the application; thus, KEKs may be exported. To export a KEK, the KEK must should be covered by another KEK. This
20 is achieved by using the GKEKs or KEKs for KEKs, and the GKEK using LSV.

Selection of symmetrical key length is the second step (Block 4). The key management method supports several key lengths depending on the symmetrical block algorithm. The key length can be adjusted between the preferred range of about 40 bits and about 192 bits, depending on the PCDB programming. For a standard
25 DES and Triple DES, keys can preferably have about a 40 bit to a about 192 bit key length, programmable in 8-bit increments by the application. This allows for variable

key lengths other than the typical 40, 56, 112, and 192 bit key lengths found on the market.

5 The third step (Block 6) of symmetrical key generation can preferably be performed six ways: 1) sample the output of a random number generator to assemble the desired length DEK; 2) sample the output of the random number generator to assemble the desired length KEK; 3) perform Diffie-Hellman g^{xy} exponential in order to arrive at a shared secret value, such as based on ANSI X9.42; 4) derive a symmetrical secret key by hashing an application supplied password or passphrase; 5) transform a key using a combination of hashing, mixing with fixed data and re-
10 hashing, XORing, etc.; or 6) import a RED key provided by the application.

15 The fourth step (Block 8) involves representing the secret key in one of preferably three ways: 1) inter-operable external form; 2) IRE (the encryption chip manufacturer) external form; or 3) IRE internal form. Numbers 2 and 3 are used to enforce the security policy, and Number 1 is used to allow shared key material with any other vendor's implementations.

20 The symmetrical key inter-operable external representation step should be used when an application chooses to exchange the chip manufacturer's secret key with another crypto vendor. The secret key should be converted from the chip manufacturer's storage format into one that is more easily extractable so that it can inter-operate with other crypto vendors.

25 If the user chooses to export a secret key, the key should be put back into an external form. The external form includes all the original random key bits, salt bits, attributes, and SHA-1 message digest. The entire external form is encrypted, including the SHA-1 message digest. The salt bits and attributes are preferably prepended to the key bits to provide randomness to skew the external key once it is encrypted in CBC mode.

The user can choose to represent the symmetrical key in the chip manufacturer's internal representation. When a secret key is loaded into the user's system, it must be put into an internal form. The internal form consists of all of the original random key bits. However, if the secret key is not a standard 56 or 192 bit key (for example, a 168 bit key), then the extracted random key bits are weakened.

If the user selects an asymmetric type of key, there are preferably three types of symmetrical keys supported: 1) Diffie-Hellman public keys for generation of a negotiated key based on a previously generated modulus base; 2) RSA public keys for both encryption and digital signatures; and 3) DSA public keys for digital signatures only. The step of key access is automatically selected with the choice of key type.

As in all public key schemes, Diffie-Hellman uses a pair of keys, public and private. However, Diffie-Hellman is unique in that it does not perform encryption/decryption or signatures as do the other public key systems. It implements a means to generate a shared secret or a negotiated DEK (i.e. traffic key or session key). The modulus, public key and private keys can be exported/imported to and from the key management scheme. The DEK can be exported, but it should be covered by one of the key maintenance KEKs (LSV, GKEKs, or an application generated KEK).

RSA uses a pair of keys, public and private, to encrypt/decrypt and implement digital signatures. However, unlike Diffie-Hellman, the modulus cannot be fixed, and a new modulus must be generated for each new public key pair. The key pair can be exported from the key maintenance scheme. The private key is returned covered by one of the key maintenance fixed KEKs or an application generated KEK.

The DSA scheme uses a pair of keys, public and private, to implement digital signatures only. Unlike RSA, the DSA fixed moduli (p and q) and generator g may be shared among a community of users. The DSA key pair can be exported from the key

maintenance. The private key is returned covered by one of the key maintenance fixed KEKs or an application generated KEK.

5 Selection of asymmetric key lengths is the next step (Block 4). The key management method supports several public key algorithms, each of which has a different requirement. The method supports keys and moduli in the preferred range of about 512 bits to about 2048 bits in multiples of 64 bits.

10 The third step (Block 6), asymmetric key generation, preferably uses the Rabin-Miller algorithm. This step generates random numbers that have a high probability of being prime. Furthermore, the method will never reuse the same modulus for an RSA or DSA new public key. The user can chose from DSA key generation, RSA key generation, and Diffie-Hellman key generation.

15 The application program has several ways to generate the public keyset for DSA operations. It can generate the p and q of the modulus in a safe or weak manner. The safe method is based on Appendix A of the X9.30 specification. It preferably goes through a formal method of creating p and q from a 160 bit seed and 16 bit counter. In this method, the application is insured of not using a cooked modulus. In the weak method, the p and q of the modulus are generated in a simple method of finding q a prime factor of p-1. This does not provide for a seed and counter but generates p and q faster.

20 The RSA key generation option requires that the RSA moduli, p and q, are generated by first finding a prime number p and then another prime number q that is close in value to p.

25 The application has several ways to generate the public keyset for Diffie-Hellman operations. It can generate the modulus p in a safe or weak manner. The safe method finds a safe prime, one of $2q+1$. In the weak method, the modulus p is generated as a random number. In this case, the modulus is weak but will be generated relatively quickly. Furthermore, the modulus p is not tested for primality.

All three of the public key generation methods described above preferably require that in either case, p and q are tested to determine if they are prime. They are tested using the Rabin-Miller primality test, and the application can specify the number of times it wants to run the test. The more times it runs, the better the probability that the number is prime. However, the more times one runs the test, the longer it takes to generate p and q . Also, as part of the primality test the prime number to be tested goes through a small divisor test of primes between 1 and 257.

The fourth step (Block 8) in the process involves representing the private key of a public key pair. The private key may be represented in one of preferably two forms: 1) inter-operable external form; and 2) IRE (the chip manufacturer) internal form.

The asymmetric key inter-operable external representation step is preferred to be used when an application chooses to exchange a chip manufacturer's private key with another crypto vendor. In this step, one must move the private key from the chip manufacturer's storage format into one that is more easily extractable so that it can inter-operate with other crypto vendors.

The user can also choose to represent the asymmetric key in the chip manufacturer's internal representation. When an application chooses to create the chip manufacturer's private key for local storage, this format is preferably used.

Section II of the Detailed Description

The key management scheme of the present invention is particularly suited for use in the cryptographic co-processor mentioned previously. This co-processor is also referred to herein by the trademark CryptIC. Also used here is the term "IRE", which stands for Information Resource Engineering, Inc., the cryptographic co-processor manufacturer and owner and assignee of the present invention and the co-processor.

The CryptIC chip provides a unique key management scheme that allows generation of several types of keys, key usage policy, and unlimited off-chip secure storage for both symmetrical and asymmetrical keys.

The user of the CryptIC chip is presented with many key generation options.

5 For symmetrical keys the user is provided with a means to create keys using the chip's randomizer, pass-phrases from the user, results from a Diffie-Hellman key exchange, importation of foreign keys (i.e. simple blobs from other Microsoft CryptoAPI™ cryptographic service providers (CSPs)), and specially created IRE Hash Message Authentication Code (HMAC) keys. The unique point with symmetrical keys is the

10 support for key generation lengths preferably between about 32 and about 192 bits, in increments of 8 bits.

For asymmetrical keys, the chip can create keys for DSA, Diffie-Hellman, and RSA using a prime number generator based on the Rabin-Miller and lowest prime divisor methods. Also, it supports some unique DSA prime generation by further

15 extending the X9.30 method. It allows the user to pass in a random seed and a means to pre-set the seed counter to something other than 0. Furthermore, asymmetrical keys can have lengths preferably between about 512 and about 2048 bits, with increments of 64 bits.

Each type of key, asymmetrical and symmetrical, includes a set of

20 programmable attributes that provide the user with a unique set of security policies. The security policies are enforced by the secure kernel embedded within the chip. The security policies allow the user to specify the type of key, the key's usage, and the key's trust level. More importantly, the key usage and key trust level are embedded as part of the secure storage of the key using a unique mix of symmetrical encryption

25 and one-way HASH authentication. This will be described later. Therefore, these key attributes are securely stored and are tamper proof, thus providing a means for secure policy decisions.

The type of key specifies what algorithm the key can be used in. Therefore, a key of type DES can not be used as triple DES or RL5 keys. This means a stronger key could not be used in a weaker manner, thus enforcing export law.

5 The key usage attribute is used to specify how a key can be used. Again, the secure kernel enforces these policy decisions; therefore, the user can not affect the policy without changing the attributes. To change the attributes, it would require the user to present the KEK the key is covered under, thus authenticating it as the owner. In any event, the key usage bits allow a user to indicate the key is a LSV, GKEK, KEK, K or KKEK.

10 The LSV is provided as the root key for providing off-chip secure key storage. The LSV can not be exported from the chip in RED or BLACK form. It can only be used to cover GKEKs. The GKEKs are used to provide safe secure storage of user keys; it is a key encryption key and it, too, can not be exported in the RED. Conventional KEKs are key encryption keys that the user can use to make their own
15 secure storage hierarchy if need be or to implement older key management protocols like X9.17. The K type is used to indicate a conventional traffic encryption key. Ks can only be used in encrypting data, not other keys. KEKs can only be used to cover other keys not data encryption. These powerful policies are enforced by the secure kernel embedded in the chip.

20 A KKEK is a key encryption key used to uncover BLACK Ks loaded through the external high-speed interface. To use the KKEK, the application loads the KKEK into the special hardware (HW) KEK register of the encryption block. Once the KKEK is loaded into the HW register, the applicant can load BLACK keys into the encryption block through the external high-speed interface. Thus allowing the
25 external high-speed interface to be used for encryption without ever exposing keys in the RED.

The key trust levels are provided to specify what types of keys can be stored under a particular KEK and the rules of exporting a particular key. There are preferably only two trust levels, trusted or untrusted. A trusted KEK can store both trusted and untrusted keys and KEKs under it. However, once a key or KEK is stored under a trusted KEK, it can no longer be moved from this KEK to another KEK. In other words, no other KEK can take ownership of the key or KEK. Any untrusted K or KEK can be stored under an untrusted KEK. Any untrusted K or KEK under the untrusted KEK can be moved under another trusted or untrusted KEK. This is because an untrusted key was created in some untrusted manner, so it is possible some outsider knows or can determine the key, so storage of the key is untrusted. Untrusted key trees can allow for greater flexibility because of the key movement that is allowed with it.

For example, the user can build applications like key escrow or key back up schemes very easily using an untrusted key tree, whereas keys under a trusted key tree are highly secured. The trusted KEK was created internally by the secure kernel and can not be reproduced or determined by the user; in fact, it will never leave the chip in the RED form. Therefore, the trusted key tree is more rigid and constrained than the untrusted tree but at the same time provides highly secure storage. As one can see, the trust levels are a powerful concept that allows the user lots of flexibility at the risk of less secure storage or a more constrained tree with increased security storage.

Further making the CryptIC's key management scheme unique is its ability for the user to store unlimited keys off-chip in any medium the user desires. To do this the chip must prepare the key in a form to allow for secure storage of it. Not only does the key need to be put in a secure form to prevent the disclosure of RED material, but it also must protect the secure attributes used to specify the key's secure policy requirements. The secure storage form must prevent these attributes from being attacked. Once these are programmed, they must not change. To do all of this, the chip provides data confidentiality and integrity.

To support data confidentiality and integrity, a special key form or IRE key blob object has been created. The IRE key blob object provides storage for a symmetrical key and consists of the following bits in this order:

5	SALT:	64 bits
	ATTRIBUTES:	32 bits
	KEY:	256 bits
	MD:	160 bits
<hr/>		
	512 bits or 64 bytes	

10 The SALT bits are just 64 bits of random data used to help provide better encryption. This random data allows the keys to be covered more securely in Cypher Block Chaining (CBC) mode with a fixed initialization vector (IV).

 The ATTRIBUTE bits are the type, use, and trust levels (i.e. GKEK, KEK, etc) that were described earlier.

15 The KEY bits are the RED key material.

 The MD bits make up the one-way HASH message digest, an SHA digest.

 To provide confidentiality, the chip encrypts the entire 512 bits using a TDES KEK in CBC mode. To provide integrity it runs a SHA one-way HASH over the SALT, ATTRIBUTES, and KEY bits and places the message digest (MD) at the end
20 before it is encrypted with a KEK in TDES. Then this BLACK key is given to the user to store or transfer as it desires.

 When a BLACK key is loaded into the device, the chip first decrypts the IRE key blob and then runs the SHA one-way HASH over the SALT, ATTRIBUTES, and KEY bits. If the message digest matches the one in the IRE key blob, then the key is
25 accepted along with its ATTRIBUTES.

As discussed earlier, the device supports trusted or untrusted key storage. In the trusted key storage the IRE key blobs are stored under a GKEK root key. Therefore, the GKEKs must be exported. They are exported under the chip's unique storage variable or KEK, the LSV. The LSV is used as the KEK covering the GKEK and the GKEKs are used to cover the application's keys. Therefore, both the GKEK and LSV can not be determined so the application's keys can not be uncovered and one has trusted storage.

Furthermore, the GKEKs are provided as another layer of security in that they protect the LSV from being attacked with known plaintext or chosen plaintext. Therefore, if the LSV was broken, all keys could be found. Now, with this layering, only the key tree under the GKEK could be broken and exposed, not the other key trees.

In summary, the cryptographic co-processor provides a flexible yet powerfully secure key management scheme that allows for a user controlled key policy manager and more importantly, unlimited off-chip secure store of symmetrical and asymmetrical key material, CGX command. However, if some sort of severe error or hardware failure occurred, the CGX kernel will reset the processor, thus transitioning to the reset state.

Section III of the Detailed Description

A detailed description of the key management scheme of the present invention taken from a programmer's guide prepared by the assignee and owner of the invention, Information Resources Engineering, Inc. (IRE) has been previously described, and reference should be made to such section for greater details.

A computer program showing the key management scheme of the present invention is provided herewith and is incorporated herein as part of the disclosure of the invention.

Although illustrative embodiments of the present invention have been described with reference to the accompanying drawings, it is to be understood that the invention is not limited to those precise embodiments, and that various other changes and modifications may be effected by one skilled in the art without departing from the scope or spirit of the invention.

Abstract of the Disclosure

A key management scheme for managing encryption keys in a cryptographic co-processor includes the first step of selecting a key from one of a symmetrical key type and an asymmetrical key type. Then, the key bit length is selected. The key is then generated and, lastly, the key is represented in either an external form or an internal form.

WHAT IS CLAIMED IS:

1. A cryptographic co-processor comprising:
a processing unit for processing data;
a read only memory electronically linked to the processing unit and
including a masked programmed cryptographic library of encryption algorithms;
and
an encryption processor for encrypting data, the encryption
processor and the processing unit being situated on the same platform.
2. A cryptographic co-processor comprising:
a digital signal processor having a memory associated therewith;
a cryptographic library having user selectable encryption algorithms
masked programmed into the memory; and
security hardware embedded within the digital signal processor.
3. A cryptographic co-processor as defined by Claim 2, wherein the
security hardware includes an encryption circuit, the encryption circuit
electronically linked to the digital signal processor and performing encryption and
decryption functions.
4. A cryptographic co-processor as defined by Claim 2, wherein the
security hardware includes a HASH circuit, the HASH circuit being electronically
linked to the digital signal processor and performing HASH functions.
5. A cryptographic co-processor as defined by Claim 2, wherein the
security hardware includes a public key accelerator circuit, the public key
accelerator circuit being electronically linked to the digital signal processor and
performing arithmetic functions.
6. A cryptographic co-processor as defined by Claim 2, wherein the
security hardware includes a random number generator, the random number

generator being electronically linked to the digital signal processor and generating random numbers used for encryption purposes.

7. A cryptographic co-processor as defined by Claim 1, which further comprises:

a laser trimmed memory, the laser trimmed memory being electronically linked to the processing unit and having stored therein a master key used by the cryptographic co-processor, the master key being programmed into the memory by laser trimming.

8. A cryptographic service software embodied in at least one of a hard disc, a floppy disc and a read-only memory (ROM), the cryptographic service software electronically communicating and being compatible with a standard operating system of a computer, the operating system having an application space and a kernel space, the cryptographic service software performing cryptographic services at the kernel space of the operating system, which comprises:

a generic layer including a kernel space level program interface; and
a cryptographic service module having a library of encryption algorithms, the module electronically communicating and cooperating with the program interface.

9. In combination, a first cryptographic service software embodied in at least one of a floppy disc and a read only memory (ROM), the first cryptographic service software electronically communicating and being compatible with a standard operating system of a computer, the operating system having an application space and a kernel space, the cryptographic service software performing cryptographic services at the application space of the operating system, the cryptographic service software comprising an application program interface, and a first cryptographic service module, the first cryptographic service module having a library of encryption algorithms, the first cryptographic service module electronically communicating and cooperating with the application program interface; and

a second cryptographic service software embodied in at least one of a floppy disc and a read only memory (ROM), the second cryptographic service software electronically communicating and being compatible with the operating system of the computer, the second cryptographic service software performing cryptographic services at the kernel space of the operating system, the second cryptographic service software including a kernel space level program interface, and a second cryptographic service module, the second cryptographic service module having a library of encryption algorithms, the second cryptographic service module electronically communicating and cooperating with the kernel space level program interface.

10. A computer having an operating system, the operating system including an application space and a kernel space, which comprises:
at least one security enabled kernel engine situated in the kernel space of the operating system;

5 cryptographic service software, the cryptographic service software being situated at least in the kernel space of the operating system, the cryptographic service software including at least one program interface electronically communicating with the at least security enabled kernel engine, and at least one
10 cryptographic service module electronically communicating with the at least one program interface, the at least one cryptographic service module including a library of encryption algorithms.

11. A computer as defined by Claim 10, which further comprises:
a cryptographic co-processor, the cryptographic co-processor including
a memory and a second library of encryption algorithms mask-programmed into the
15 memory, the co-processor electronically communicating with the at least one program interface of the cryptographic service software.

12. A method of securely communicating between an application program and a secure kernel of an integrated circuit, the secure kernel having stored therein cryptographic algorithms, the integrated circuit further having a register, a command
20 block memory and a kernel block memory, which comprises the steps of:

storing in the register of the integrated circuit the address of the kernel block memory;

transferring the kernel block memory address stored in the register to the secure kernel;

25 reading by the secure kernel the contents of the kernel block memory at the transferred address, the contents of the kernel block memory at the transferred address containing at least one pointer address corresponding to a memory location in the command block memory; and

fetching by the secure kernel the contents of the memory location of the command block memory corresponding to the at least one pointer address, the contents of the memory location including at least one of a command and an argument.

5

13. A hardware secure memory area, which comprises:

a main communication bus;

a plurality of secondary communication buses;

a plurality of bus transceivers coupling the plurality of secondary

10 communication buses to the main communication bus; and

a plurality of memory circuits coupled to the plurality of communication buses, each bus transceiver selectively isolating a secondary communication bus to which the bus transceiver is associated from the main communication bus and selectively causing communication between the associated
15 secondary communication bus and the main communication bus.

14. A hardware secure memory area, which comprises:

a main communication bus;

a first bus transceiver coupled to the main communication bus;

a second bus transceiver coupled to the main communication bus;

20 a third bus transceiver coupled to the main communication bus;

a key communication bus coupled to the first bus transceiver;

a key cache coupled to the key communication bus for writing and
reading keys;

25 a key random access memory coupled to the key communication bus
for writing and reading cryptographic operations and keys;

a processor memory for writing and reading cryptographic algorithms,
operations and keys;

an external memory communication bus coupled to the second bus
transceiver;

an external memory coupled to the external memory communication bus for writing and reading application programs and commands;
a cryptographic algorithm communication bus coupled to the third bus transceiver;
5 a scratch memory coupled to the cryptographic algorithm communication bus for writing and reading cryptographic calculations; and
a memory coupled to the cryptographic algorithm communication bus for storing cryptographic algorithms.

10 15. A hardware secure memory area, which comprises:
a main communication bus;
a plurality of bus transceivers coupled to the main communication bus for controlling access to and from the main communication bus;
a plurality of secondary communication buses coupled to the bus
15 transceivers; and
a plurality of memory circuits coupled to the plurality of secondary communication buses.

20 16. A method of expanding a protected memory area of a secure kernel into an unprotected memory area in an integrated circuit, which comprises the steps of:
initializing the secure kernel including the step of sending thereto a command signal having a starting memory block address and the number of memory blocks to be protected;
reading the command signal by the secure kernel;
25 writing the starting memory block address and the number of memory blocks into a memory of the secure kernel; and
allocating the starting memory block address and the number of memory blocks as protected memory.

17. A method of expanding a protected memory area of a secure kernel into an unprotected memory area in an integrated circuit, which comprises the steps of:

5 initializing the secure kernel by a command generated by an application program, the command causing arguments attached to the command to be vectored into the secure kernel, the arguments including pointer data and the number of protected memory blocks requested, the pointer data containing the starting memory address of the protected memory requested;

reading the arguments by the secure kernel;

10 writing into a data memory reserve register a code corresponding to the number of memory blocks to be protected; and

allocating the number of memory blocks requested to be protected as protected memory starting at the starting memory block address contained in the pointer data.

15 18. A method of expanding a secured memory into an unprotected memory to define an additional secured memory area, the secured memory being expanded to accommodate storage of an extended code, the extended code being initially stored in the unprotected memory in a location which will become the additional secured memory area, the secured memory and the unprotected memory forming parts of an integrated circuit, the integrated circuit having a serial number stored in a memory thereof, which comprises the steps of:

retrieving by an authorizing party the serial number stored in the memory of the integrated circuit and the extended code proposed to be stored in the requested expanded secured memory;

25 verifying by the authorizing party whether the extended code is acceptable to be stored in the expanded secured memory of the integrated circuit;

generating a token signal by the authorizing party and communicating the token signal to the integrated circuit, the token signal including at least the digital signature of the extended code, as computed by the authorizing party,

receiving the token signal by the integrated circuit and parsing the token signal to separate the digital signature of the authorizing party;

verifying by the integrated circuit the digital signature of the authorizing party parsed from the token signal which, if verified, indicates that the authorizing party authorizes the expansion of the secured memory by the integrated circuit; and

invoking by the integrated circuit a command to expand the secured memory so that the additional secured memory area now encompasses the location of the unprotected memory where the extended code is stored.

10 19. A method of expanding a secured memory as defined by Claim 18, wherein the token signal further includes the serial number of the integrated circuit; and wherein the method further comprises the steps of:

parsing by the integrated circuit from the token signal returned by the authorizing party the serial number of the integrated circuit; and

15 verifying by the integrated circuit the serial number parsed by the token signal by comparing the parsed serial number with the serial number stored in the non-volatile memory of the integrated circuit.

20 20. A method of expanding a secure kernel memory area into an unprotected memory, while providing security to the unprotected memory area and validating an extended code, the secure kernel memory and the unprotected memory forming part of an integrated circuit, comprising the steps of:

reading an integrated circuit serial number;

communicating the serial number to a manufacturer;

loading code into a memory;

25 loading a token into a memory, the token includes the ICs serial number;

digitally signing the extended code by a trusted authority;

copying the extended code into the ICs unprotected memory;

invoking a command to add an extension to the secure kernel memory;
specifying which memory blocks are to be acquired from the
unprotected memory area;
disabling further program access to the unprotected memory area;
5 verifying the digital signature computed over the extended code is
authentic;
locking the extended code into the protected memory area; and
permitting the secure kernel to make calls to the protected memory
area.

10 21. A method of reconfiguring the functionality of an integrated circuit, the
integrated circuit having a serial number stored in a memory thereof, which comprises
the steps of:

retrieving the serial number from the integrated circuit;
transmitting a token request signal by the integrated circuit to an
authorizing party, the token request signal including at least the serial number of the
integrated circuit;
transmitting a return token signal from the authorizing party to the
integrated circuit, the return token signal including at least the serial number of the
integrated circuit, a reconfiguration code to be used for reconfiguring the integrated
circuit and a digital signature of the authorizing party;
parsing the return token signal by the integrated circuit to extract the
serial number, reconfiguration code and digital signature;
verifying the serial number by the integrated circuit by comparing the
parsed serial number from the return token signal with the serial number stored in the
memory of the integrated circuit;
verifying the digital signature of the authorizing party by the integrated
circuit using a public key stored in the memory of the integrated circuit;
storing the parsed reconfiguration code from the return token signal in
a memory of the integrated circuit; and

reconfiguring the integrated circuit in accordance with the reconfiguration code.

22. A method of reconfiguring the functionality of an integrated circuit, the integrated circuit having a serial number stored in a memory thereof, which comprises the steps of:

retrieving the serial number from the integrated circuit;

transmitting by the integrated circuit a token request signal to an authorizing party, the token request signal including the serial number and a reconfiguration code;

transmitting a return token signal from the authorizing party to the integrated circuit, the return token signal including the serial number of the integrated circuit, the reconfiguration code and a digital signature of the authorizing party;

parsing the return token signal by the integrated circuit to extract the serial number, reconfiguration code and digital signature;

verifying the serial number by the integrated circuit by comparing the parsed serial number from the return token signal with the serial number stored in the memory of the integrated circuit;

verifying the digital signature of the authorizing party by the integrated circuit using a public key stored in a memory of the integrated circuit;

storing the reconfiguration code in a memory of the integrated circuit;

and

reconfiguring the integrated circuit in accordance with the reconfiguration code.

23. A method of generating a recovery key encryption key (RKEK) in a secure manner by an integrated circuit and a key recovery escrow agent, which comprises the steps of:

generating by the integrated circuit a first number having a private component and a public component;

generating by the escrow agent a second number having a private component and a public component;
providing the public component of the first number to the escrow agent;
providing the public component of the second number to the integrated circuit;
conducting a mathematical operation by the integrated circuit using the private component of the first number, and the public component of the second number to create the RKEK; and
conducting a mathematical operation by the escrow agent using the private component of the second number, and the public component of the first number to create the RKEK.

24. A method of generating a recovery key encryption key (RKEK) in a secure manner by an integrated circuit and a key recovery escrow agent, the integrated circuit having a unique serial number stored in a memory of the integrated circuit, which comprises the steps of:

generating by the integrated circuit a first number having a private component and a public component;
generating by the escrow agent a second number having a private component and a public component;
retrieving by a third party the serial number of the integrated circuit and comparing the serial number with a serial number stored in a memory of the third party to verify the identity of the integrated circuit;
generating by the third party a message containing at least a digital signature of the third party authorizing the generation of the RKEK and communicating the message to the integrated circuit;
providing the public component of the second number to the integrated circuit; and

conducting a Diffie-Hellman modulo-exponentiation mathematical operation by the integrated circuit using the private component of the first number, and the public component of the second number to create the RKEK.

25. A method of generating a recovery key encryption key (RKEK) as defined by Claim 24, wherein the message generated by the third party and communicated to the integrated circuit further includes the serial number of the integrated circuit, and wherein the method further comprises the step of:

verifying by the integrated circuit the accuracy of the serial number included in the message by comparing the serial number of the message with the serial number stored in the memory of the integrated circuit.

26. A method of generating a recovery key encryption key (RKEK) as defined by Claim 25, which further comprises the step of:

verifying by the integrated circuit the accuracy of the digital signature of the third party contained in the method..

27. A method of generating a recovery key encryption key (RKEK) as defined by Claim 26, which further comprises the steps of:

providing the public component of the first number to the escrow agent; and

conducting a Diffie-Hellman modulo-exponentiation mathematical operation by the escrow agent using the private component of the second number, and the public component of the first number to create the RKEK.

28. A controller circuit for switching between a user mode and a kernel mode in a processor comprising;

a processor;

a program counter electrically connected to the processor for monitoring program fetch addresses;

a kernel program fetch supervisor circuit having a predetermined address value stored within, electrically connected to the program counter for comparing the address in the program counter to the predetermined address value stored within;

a program memory electrically connected to the program counter;

a flip-flop circuit electrically connected to the kernel program fetch supervisor circuit for switching between setting a user mode bit and a kernel mode bit;

a kernel data fetch supervisor circuit electrically connected to the processor for comparing a data fetch address to a predetermined memory address range;

a data memory electrically connected to a processor data interface for storing data;

a first AND circuit coupled to the flip-flop and the kernel data fetch supervisor circuit for activating and deactivating a violation reset;

and a second AND circuit coupled to the first AND circuit and the kernel program fetch supervisor circuit for activating and deactivating the violation reset bit.

29. A method of monitoring and controlling program fetch addresses and data fetch addresses from a processor to control access to a protected memory comprising the steps of:

fetching a program opcode;

reading a program opcode address;

determining whether the program opcode address is fetched from one of a protected program memory address and an unprotected program memory address;

resetting the processor when the program opcode is fetched from the protected program memory address;

fetching a data operand when the program opcode address is fetched from the unprotected program memory address;

fetching a data operand and reading the data operand address;

determining whether the data operand address is fetched from one of a protected data memory address and an unprotected data memory address;
resetting the processor when the data operand is fetched from the protected data memory address;
calling a starting address of the protected program memory when the data operand address is fetched from the unprotected data memory;
fetching a second program opcode;
reading the second program opcode address;
determining whether the second program opcode address is fetched from one of a protected program memory address and an unprotected program memory address;
fetching a third program opcode when the second program opcode address is fetched from the unprotected memory address; and
fetching a second data operand when the second program opcode address is fetched from the protected memory address.

30. A method of managing the use of keys in cryptographic co-processor, which comprises the steps of:

selecting a key from one of a symmetrical key type and asymmetrical key type;
selecting a bit length from the selected key;
generating the key; and
representing the key in one of an external form and an internal form.

1/45

FIG-1

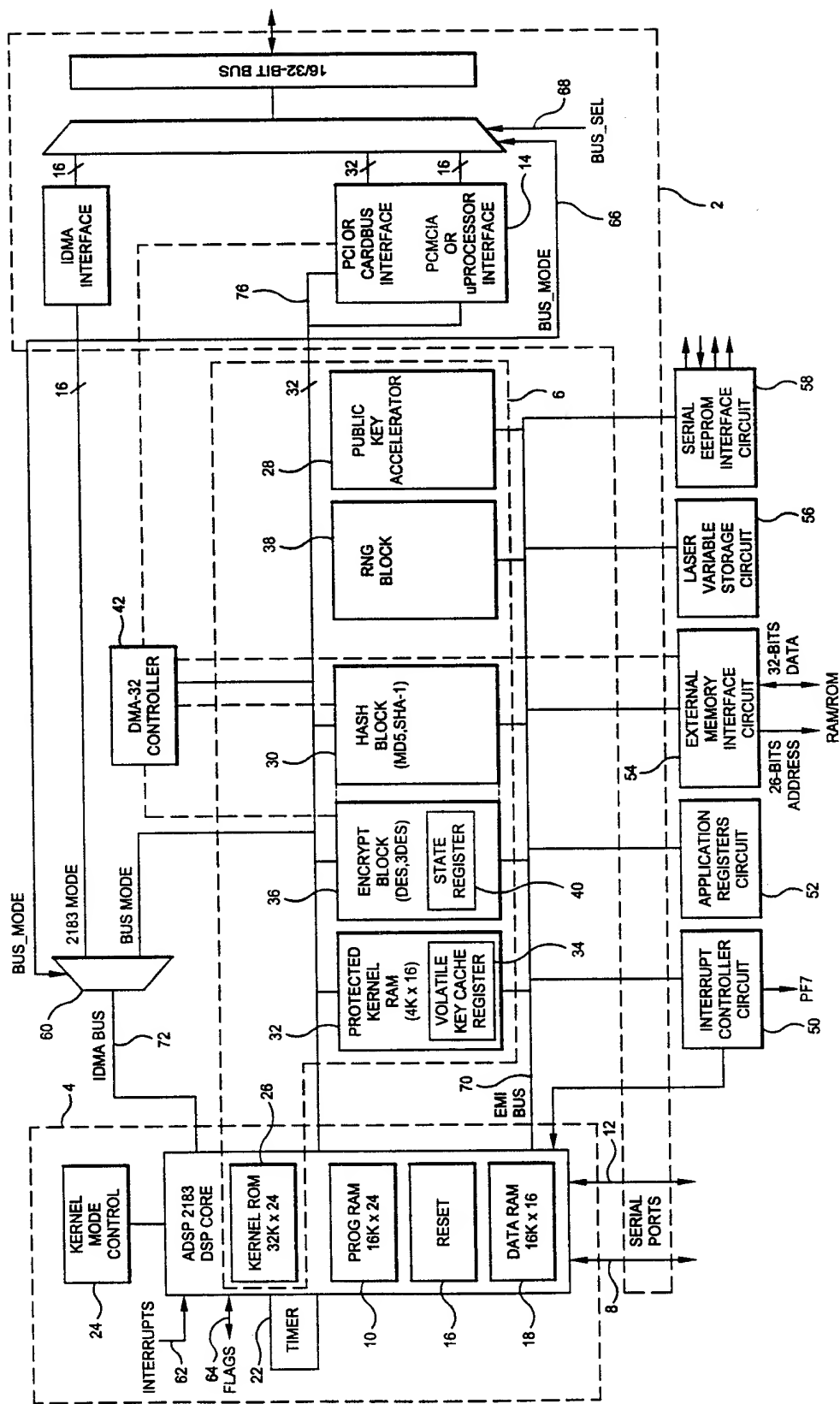


FIG-2 PROGRAM MEMORY (MMAP=0)

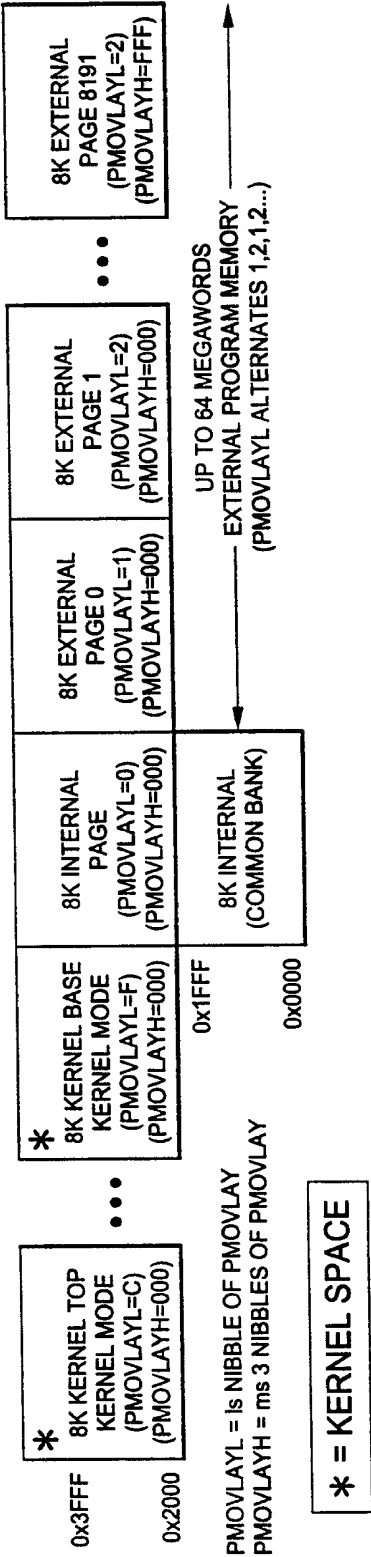


FIG-3 PROGRAM MEMORY (MMAP=1)

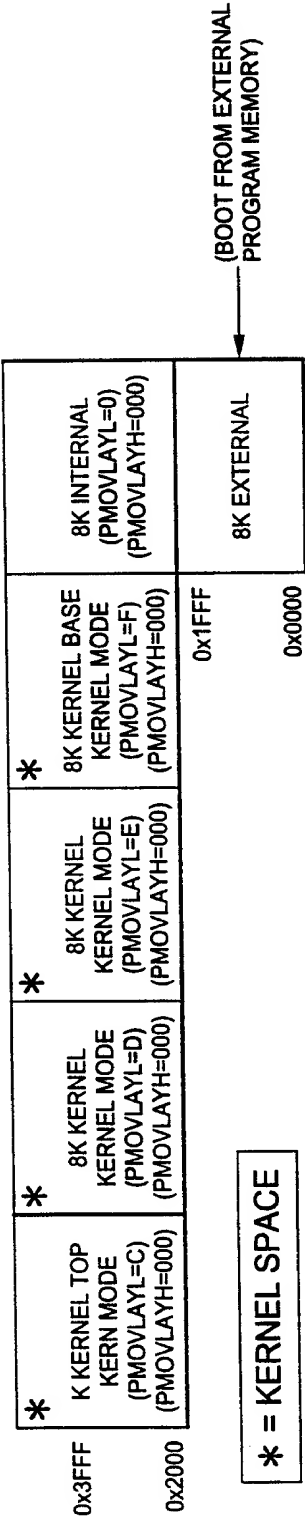


FIG-4 DATA MEMORY

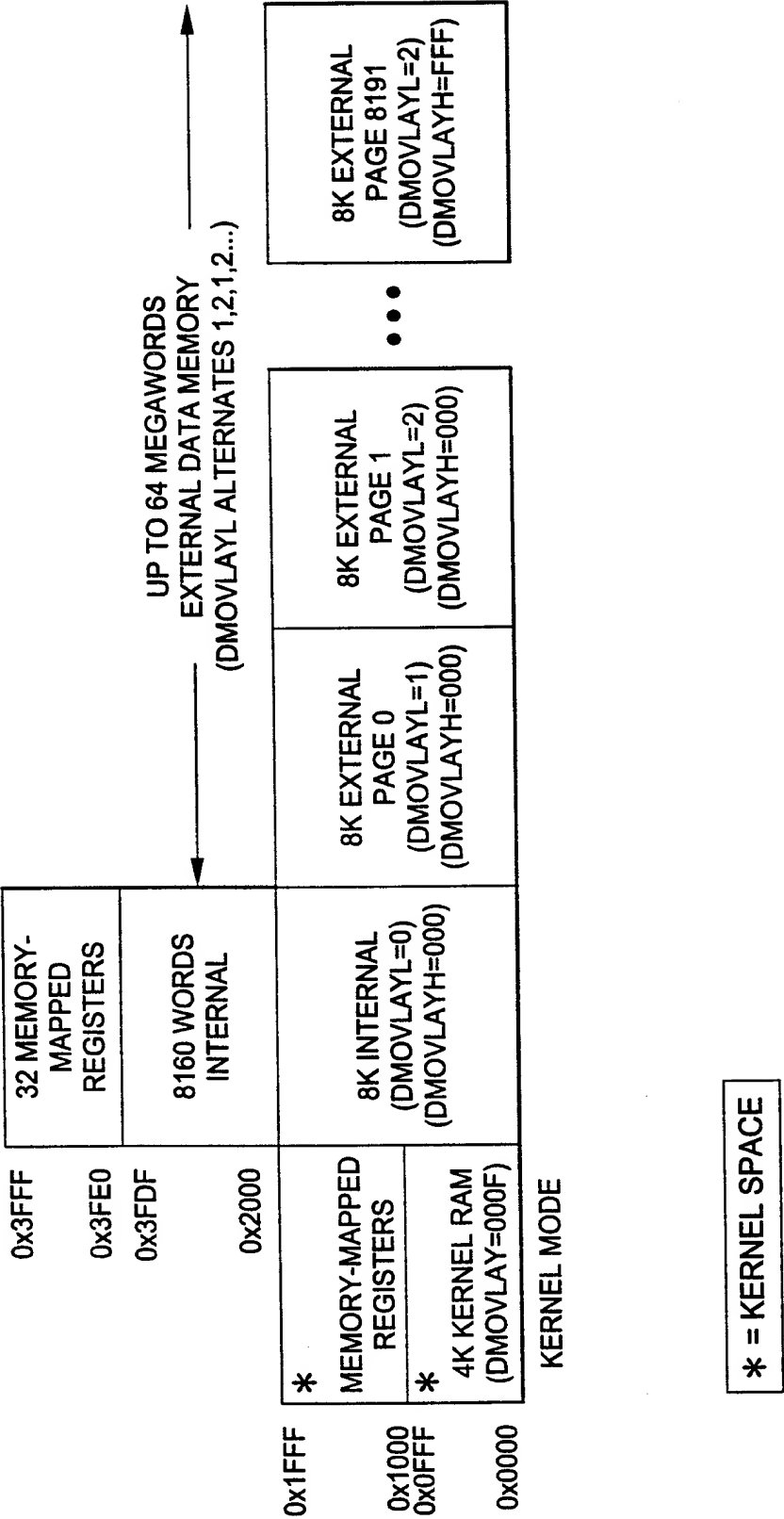
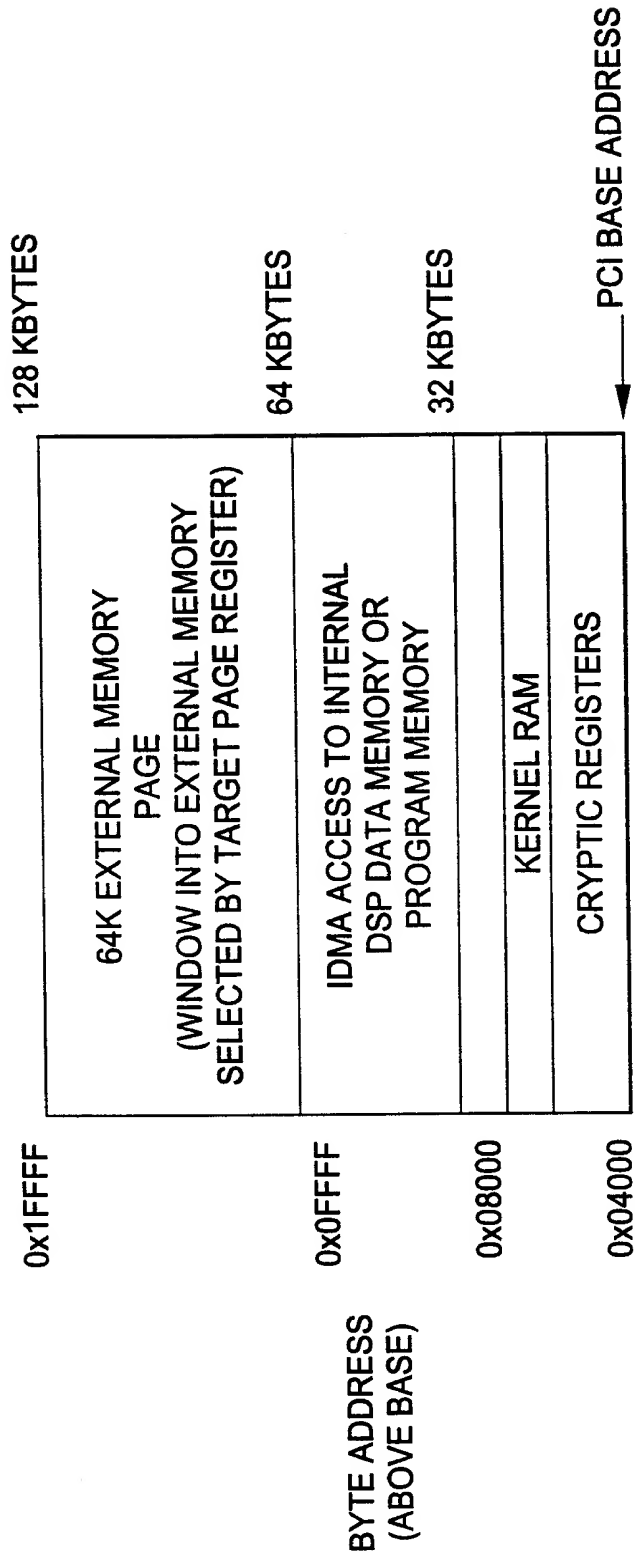


FIG-5 PCI MEMORY MAP



6/45

FIG-6 32-BIT DMA SUBSYSTEM

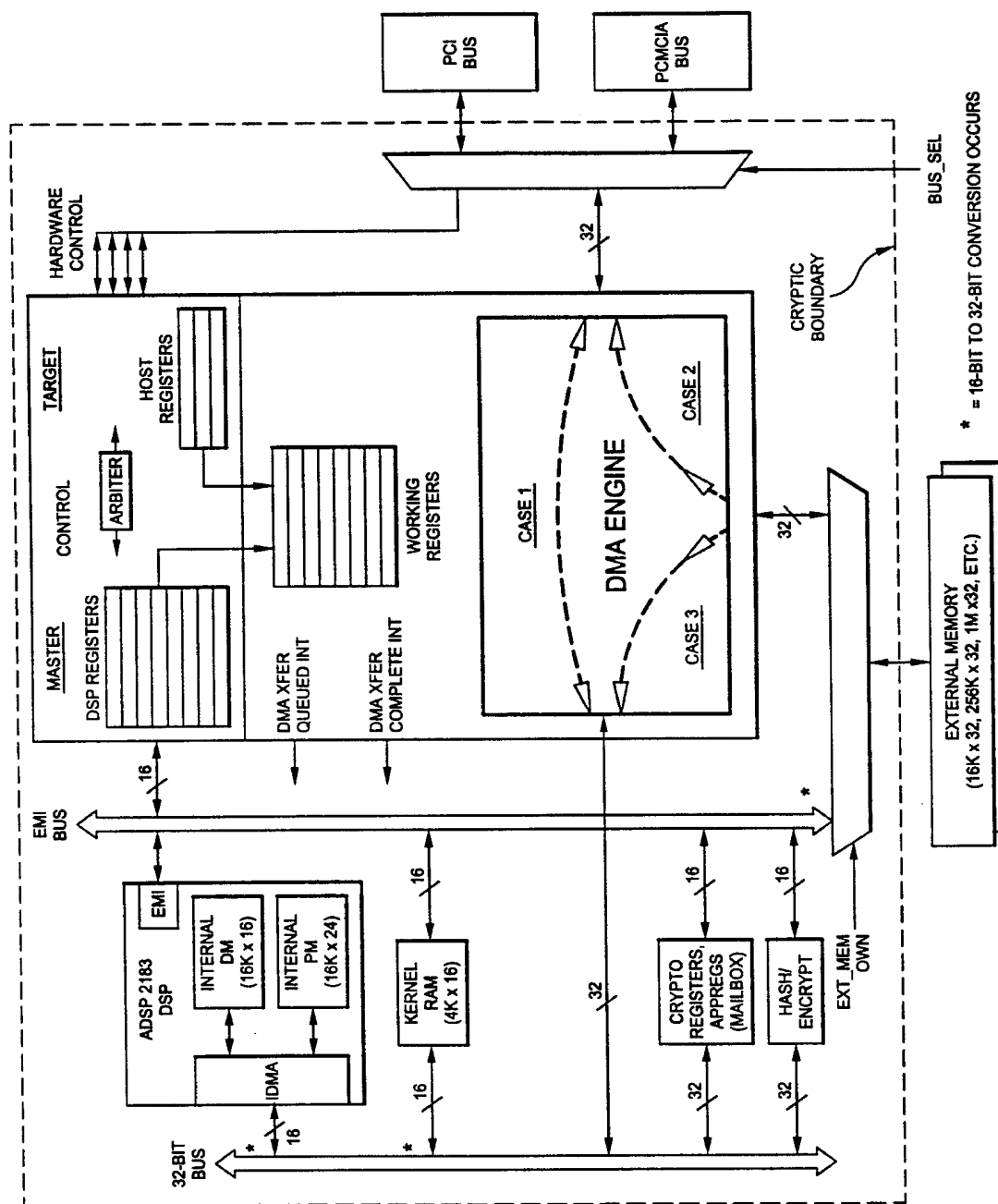
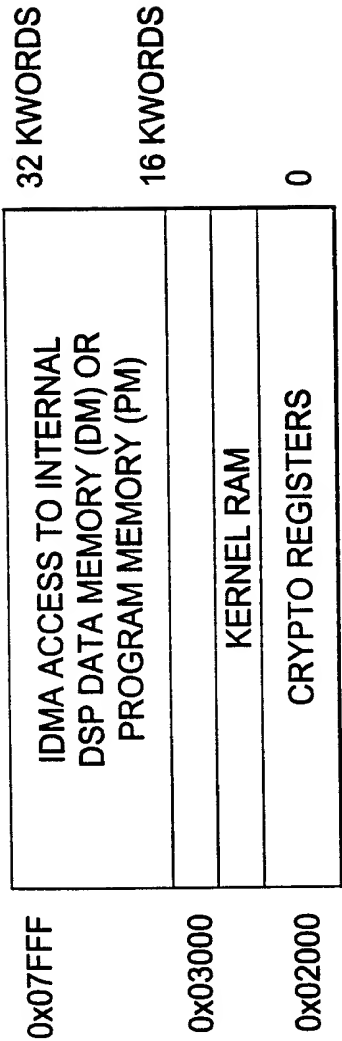
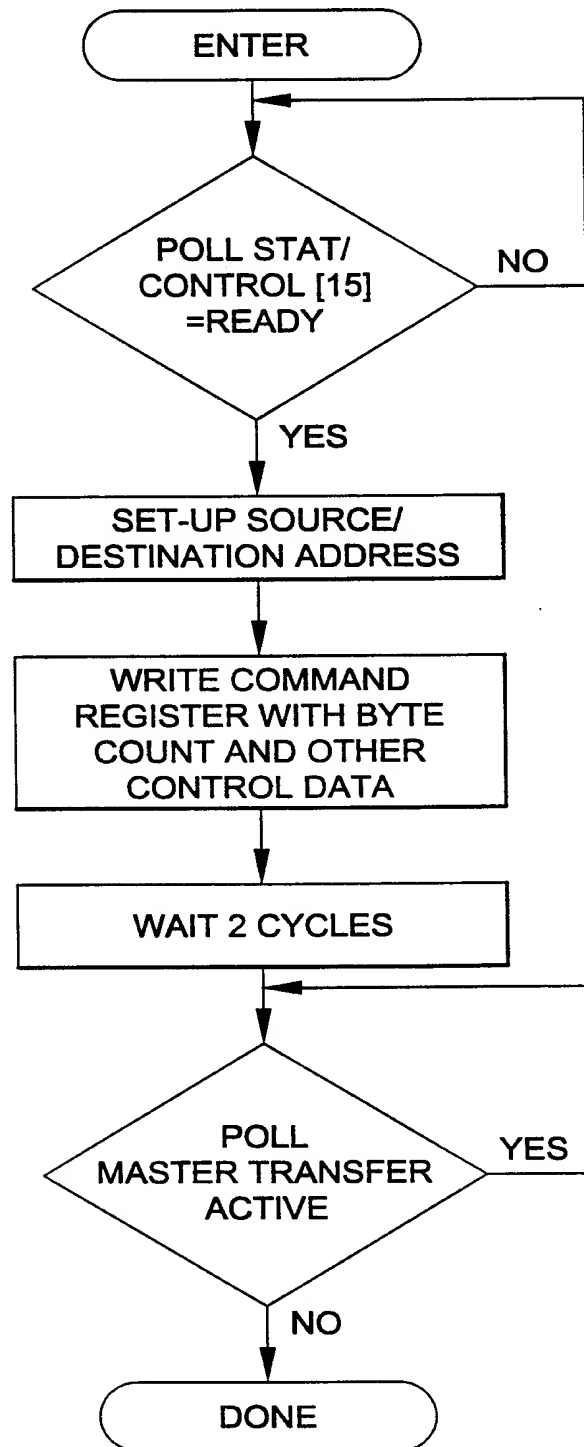


FIG-7 DSP "LOCAL" MEMORY MAP

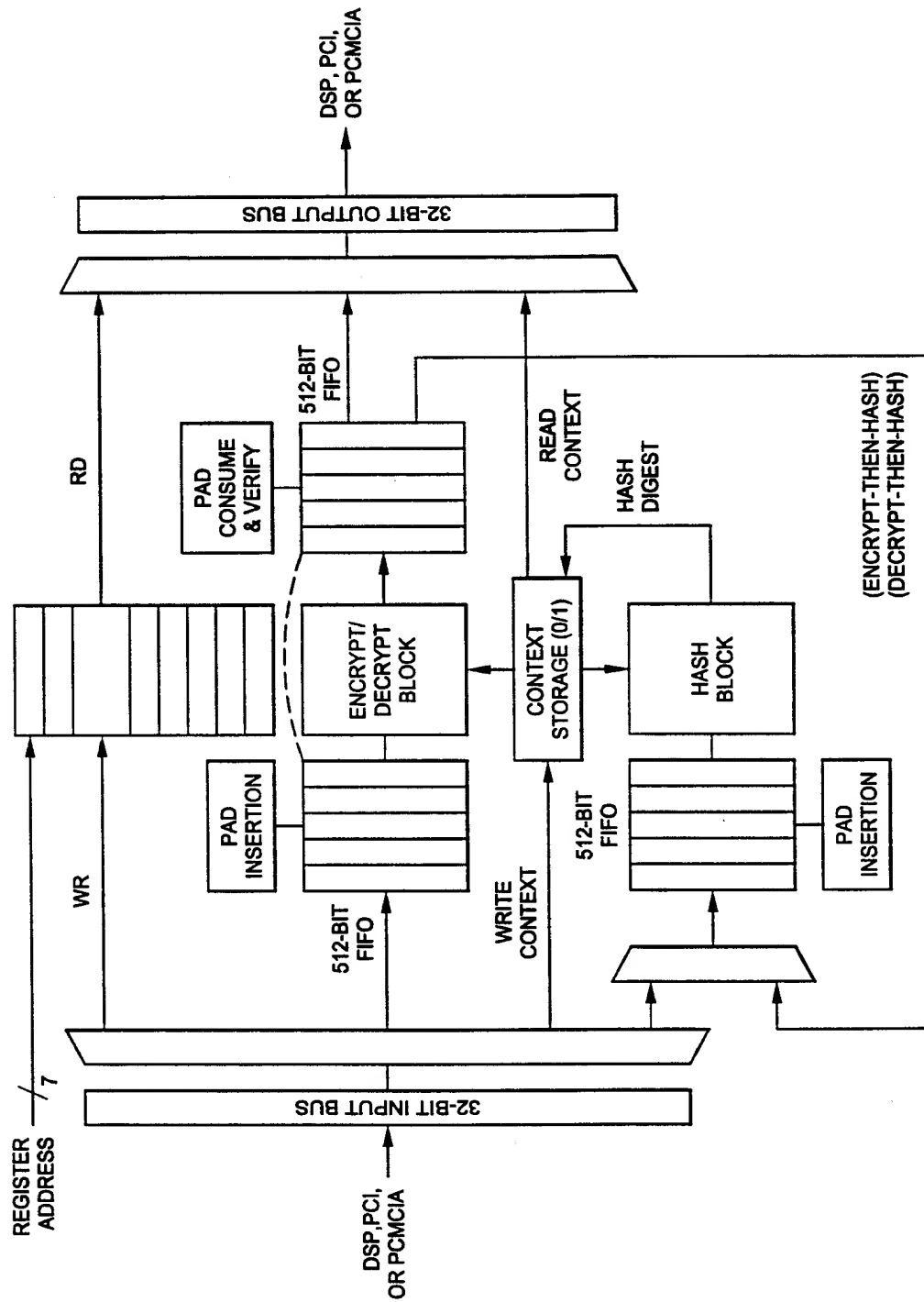


8/45

FIG-8 MASTER DMA FLOWCHART

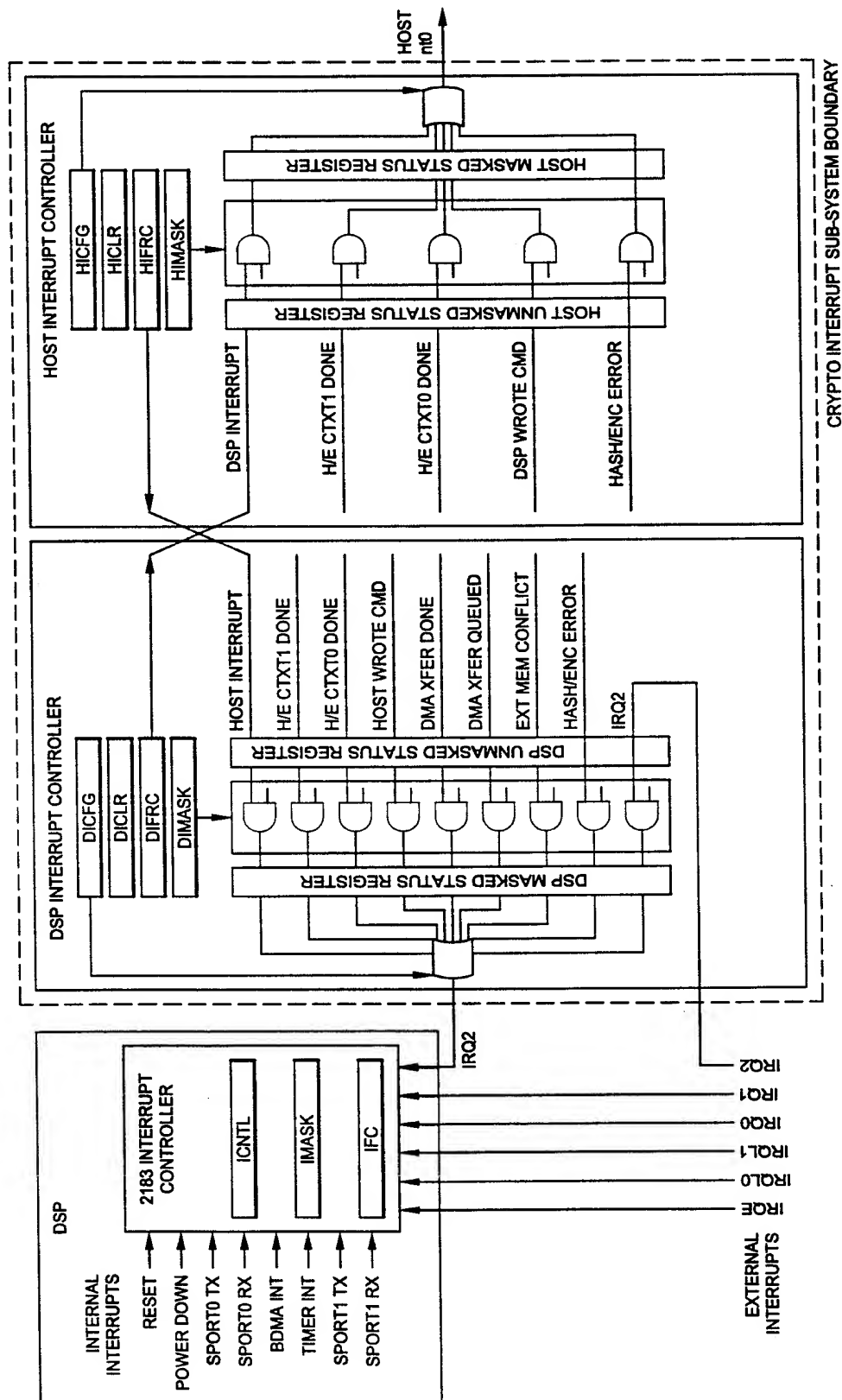
9/45

FIG-9 HASH/ENCRYPT FUNCTIONAL BLOCK DIAGRAM



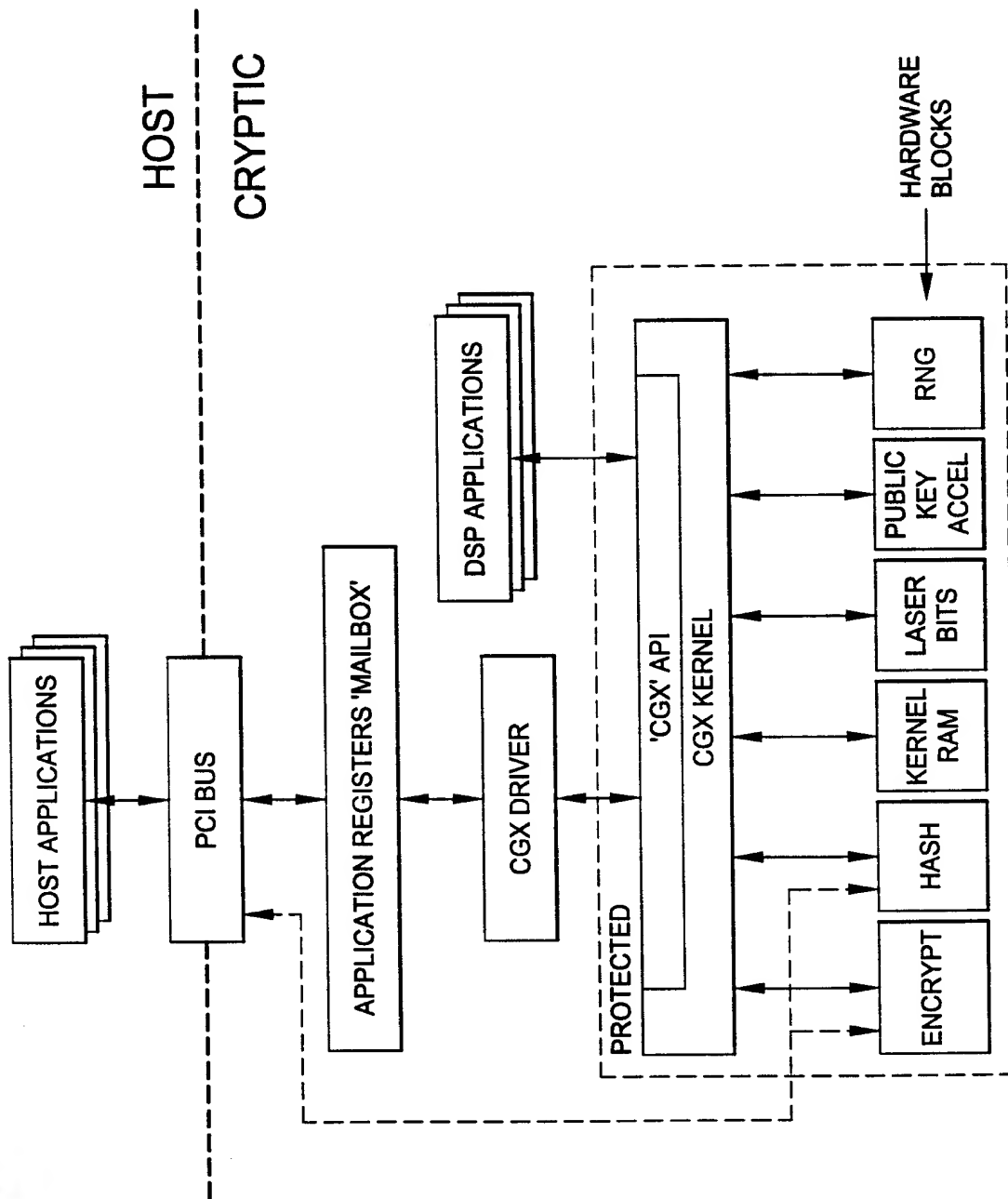
10/45

FIG-10 INTERRUPT CONTROLLER BLOCK DIAGRAM



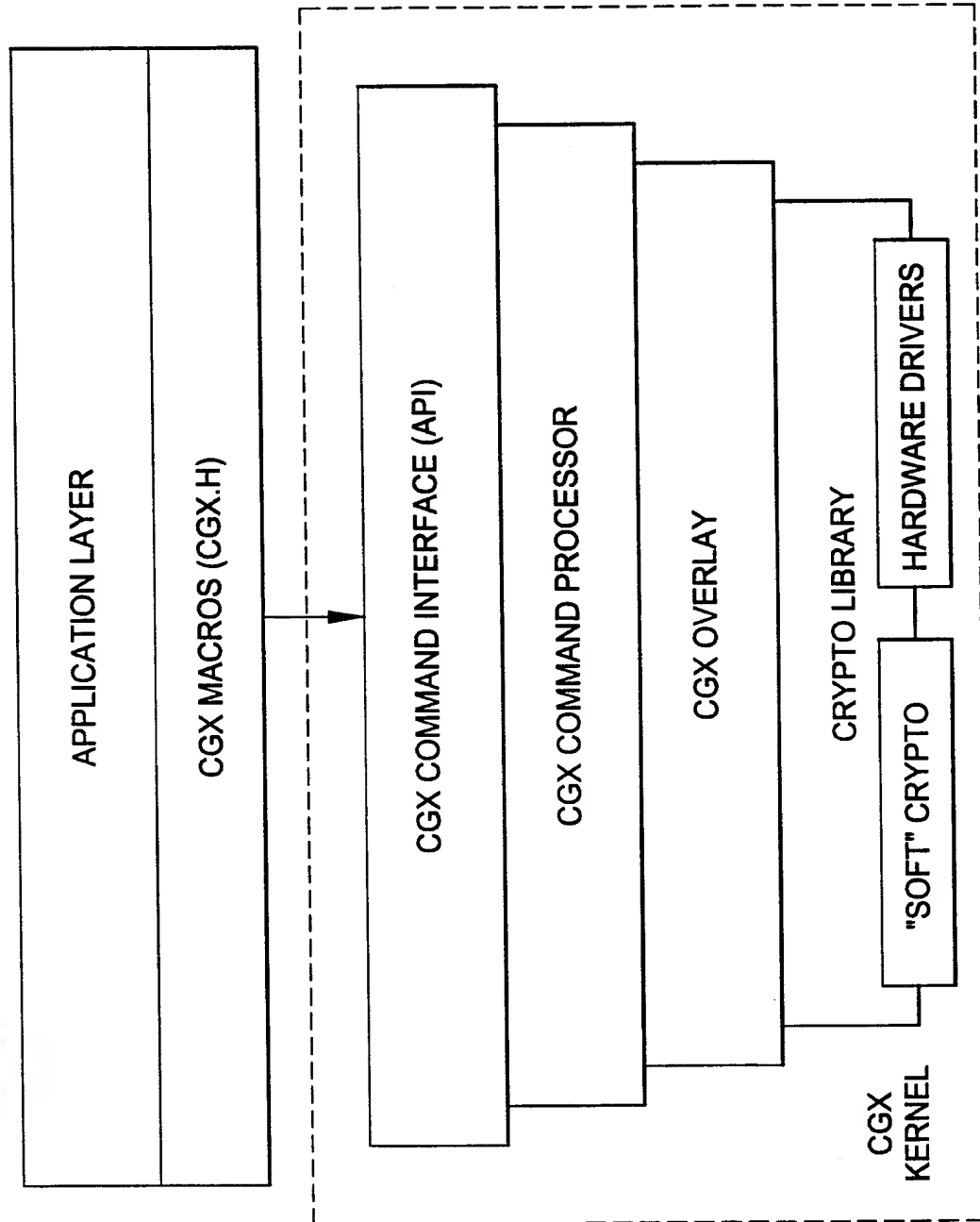
11/45

FIG-11 CGX SOFTWARE INTERFACE

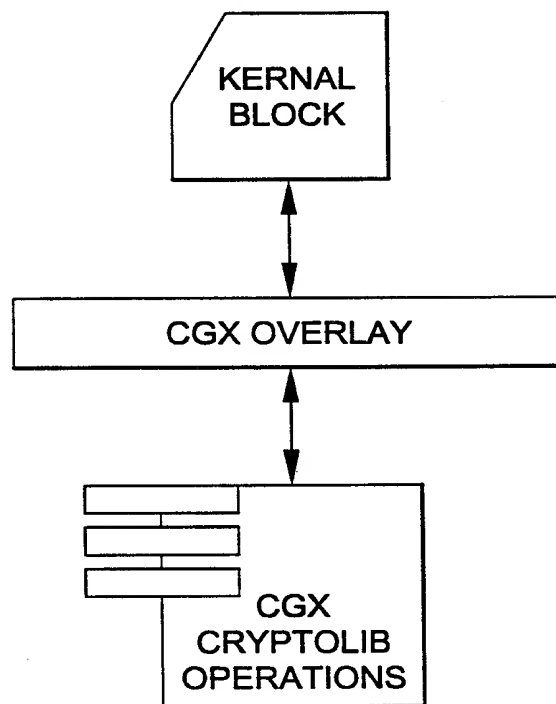
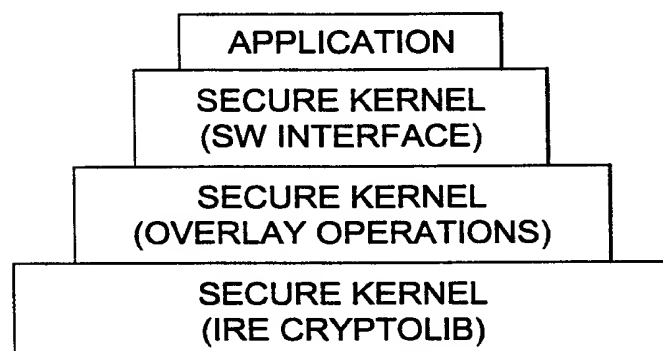


12/45

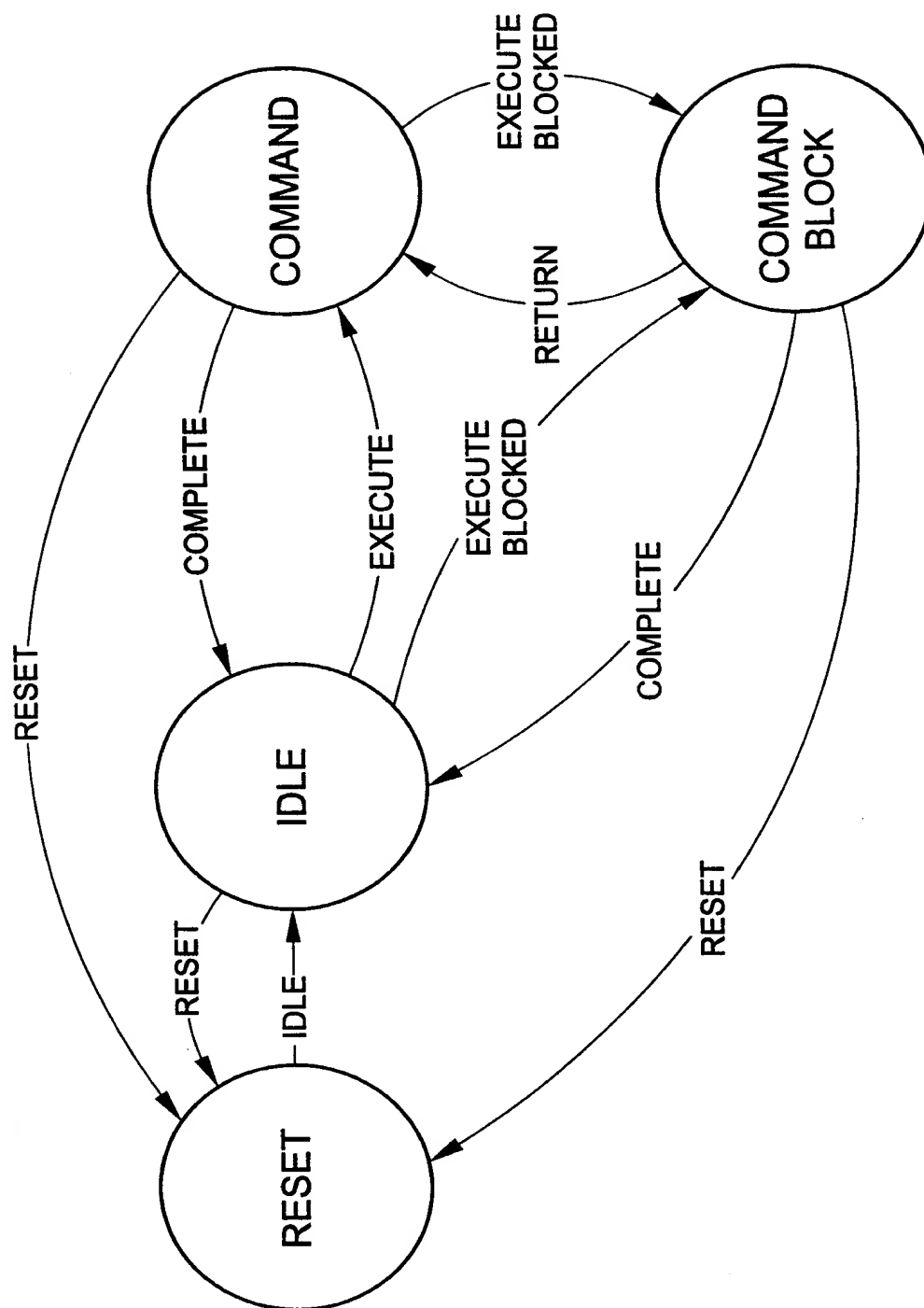
FIG-12 CRYPTIC SOFTWARE LAYERS



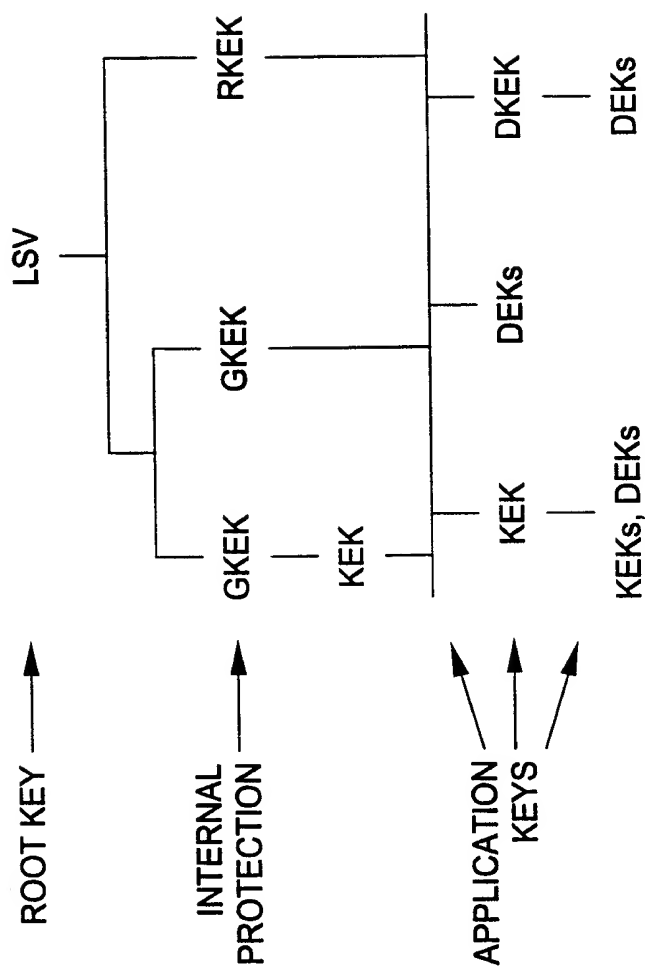
13/45

FIG-13 CGX OVERLAY INTERFACE**FIG-14** CGX KERNEL CRYPTOGRAPHIC SERVICE HIERARCHICAL INTERFACE

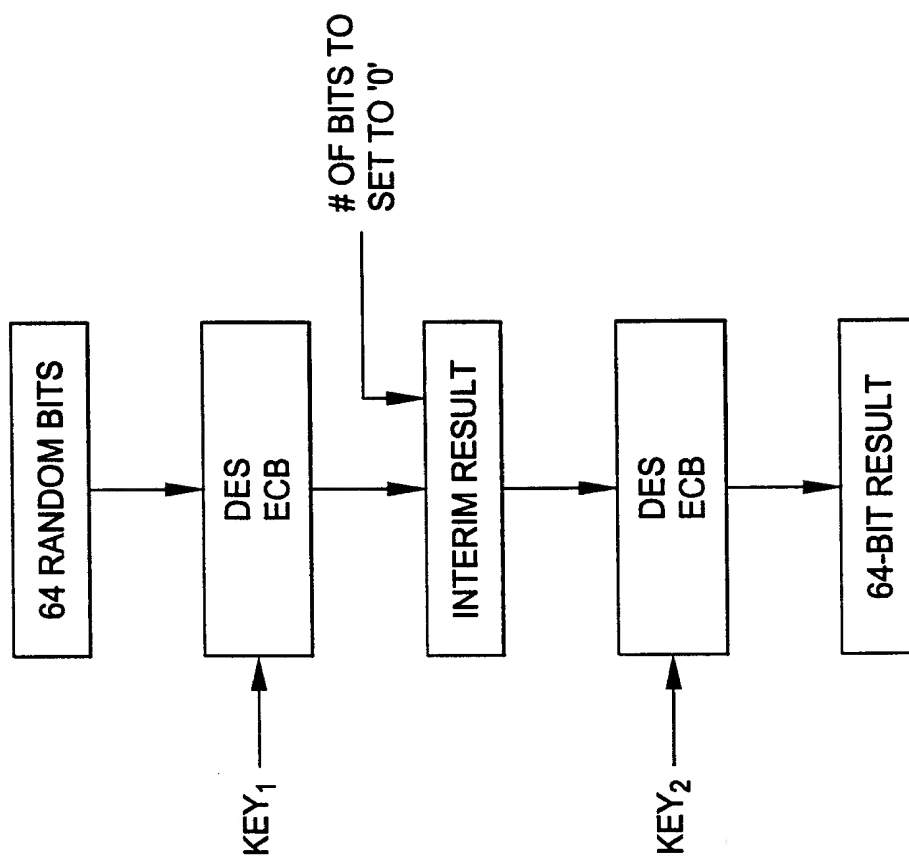
14/45

FIG-15 CGX KERNEL STATE DIAGRAM

15/45

FIG-16 KEK HEIRARCHY

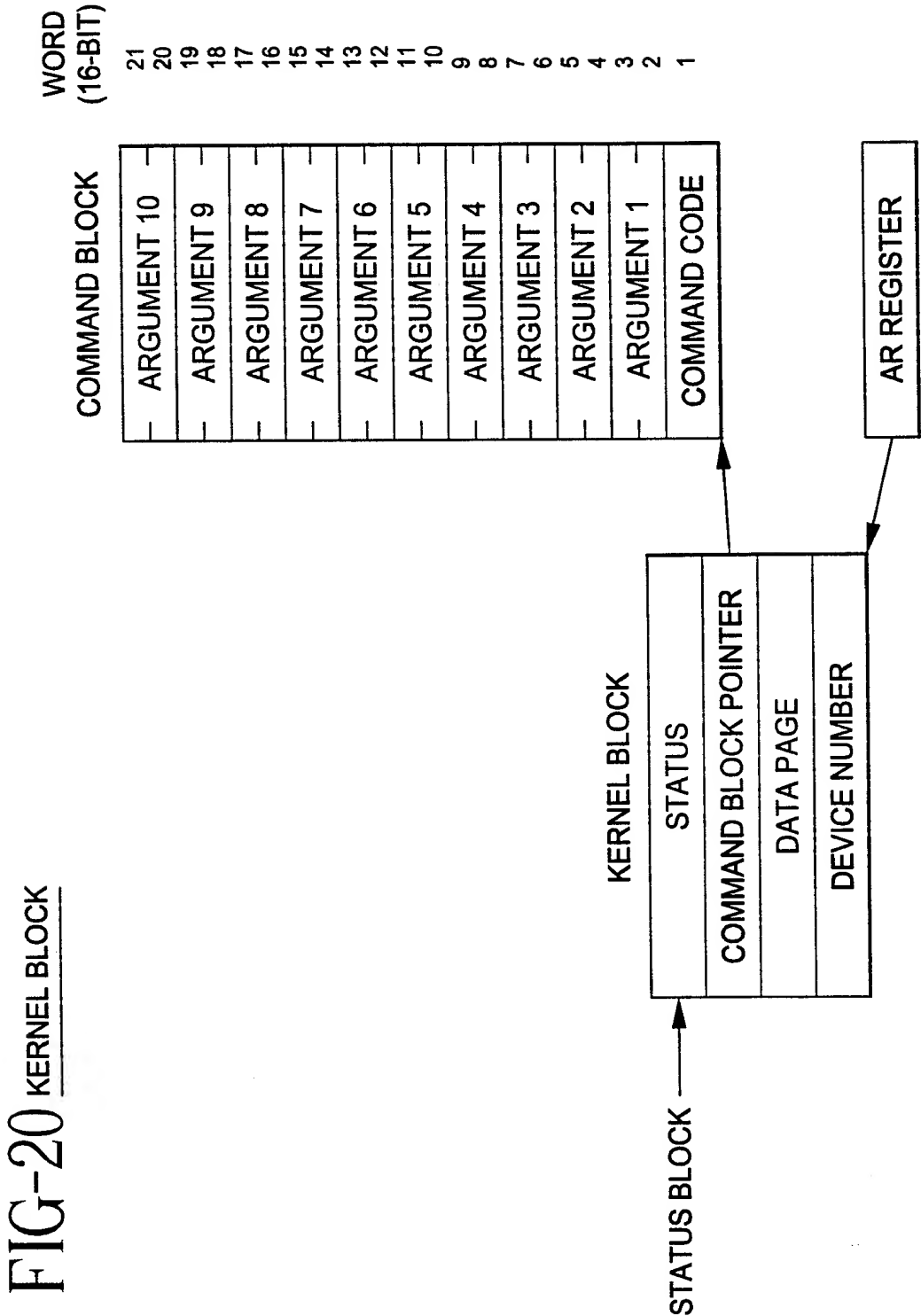
16/45

FIG-17 SYMMETRIC KEY WEAKENING ALGORITHM

18/45

FIG-19 PCDB DATA TYPE DEFINITION

```
typedef struct _token_pcdb {  
    signblock token_signature;  
    UINT16    serial_number[CGX_SERIAL_NUMBER_LEN];  
    UINT16    length;  
    UINT16    reserved;  
    UINT16    pcdb[CGX_PCDB_LEN];  
    UINT16    reserved1;  
} token_pcdb;
```

20/45

FIG-21 KERNEL BLOCK OBJECT DEFINITION

```

typedef unsigned short UINT16; /* unsigned 16 bit integer */

typedef struct _kernelblock {
    UINT16 DeviceNo; /* command class and memory model */
    UINT16 dp; /* ADI-2183 specific, data page of cmd/stat */
    cmdblock *cb; /* command block object, low memory first */
    UINT16 status; /* command execution status */
} kernelblock;

```

FIG-22 COMMAND BLOCK OBJECT DEFINITION

```

typedef (void *) VPTR; /* a pointer to any data type or object */

typedef struct _cmdblock {
    unsigned short cmd; /* micro command */
    VPTR argument[CGX_MAX_ARGS]; /* argument list */
} cmdblock;

```

21/45

FIG-23 SECRET KEY OBJECT DEFINITION

```

typedef WORD16 unsigned short;          /* 16 bits of data */

typedef struct _secretkey {
    UINT16  extra;          /* for application's use */
    UINT16  type;           /* the secret key type */
    UINT16  length;         /* key length */
    UINT16  k[CGX_RAW_SECRET_KEY_HASH_LENGTH]; /* secret key, salt, */
                                                /* plus some attributes & len */
                                                /* plus a SHA-1 HASH digest */
} secretkey;

```

FIG-24 PUBLIC KEYSET OBJECT DEFINITION

```

typedef void      *VPTR;
typedef unsigned short WORD16;

typedef struct _publickey {
    WORD16  extra;          /* 16 bits of space reserved for the application */
    WORD16  type;           /* public key algorithm; defines keyset object types */
    WORD16  length;         /* modulus length of public key (in bits) */
    VPTR    modulus;        /* modulus object */
    VPTR    pubkey;         /* public key object */
    VPTR    privkey;        /* private key object */
} publickey;

```

22/45

FIG-25 DIFFIE-HELLMAN PUBLIC KEYSET OBJECTS

```

typedef unsigned short WORD16;

#define CGX_2048_BITS 128 /* To allocate 2048 bits */
#define CGX_1024_BITS 64 /* To allocate 1024 bits */
#define CGX_64_BITS 4 /* To allocate 64 bits (salt) */

typedef UINT16 Twokbits[CGX_2048_BITS];
typedef UINT16 Onekbits[CGX_1024_BITS];
typedef UNIT16 Saltbits[CGX_64_BITS];

/* Diffie-Hellman Public Keyset Objects */
typedef struct _DHmodulus {
    WORD16 n[128]; /* max 2048-bit modulus */
    WORD16 g[128]; /* max 2048-bit base */
} DHmodulus;

typedef struct _DHpubkey {
    WORD16 Y[128]; /* max 2048-bit public key */
} DHpubkey;

typedef struct _DHprivkey {
    WORD16 salt[4]; /* salt for covering purposes */
    WORD16 y[128]; /* max 2048-bit private key */
} DHprivkey;

```

23/45

FIG-26 RSA PUBLIC KEYS OBJECTS

```

typedef unsigned short WORD16;

typedef struct _RSAm modulus {
    WORD16 n[128]; /* maximum 2048 bit modulus */
} RSAm modulus;

typedef struct _RSApubkey {
    WORD16 e[2]; /* maximum 32 bit public key */
} RSApubkey;

typedef struct _RSAprivkey {
    WORD16 salt[4] /* rndm data for covering purposes */
    WORD16 p[64]; /* 1024 bit max prime modulus */
    WORD16 q[64]; /* 1024 bit max prime modulus */
    WORD16 dP[64]; /* 1024 bit max, d mod (p-1) */
    WORD16 dQ[64]; /* 1024 bit max, d mod (q-1) */
    WORD16 qInv[64]; /* 1024 bit max, q^-1 mod p */
    WORD16 d[128]; /* maximum 2048 bit private key */
} RSAprivkey;

```

24/45

FIG-27 DSA PUBLIC KEYSET OBJECTS

```

typedef unsigned short WORD16;

typedef struct _DSAm modulus {
    WORD16 p[128]; /* maximum 2048 bit modulus */
    WORD16 q[10]; /* fixed 160 bit modulus */
    WORD16 g[128]; /* maximum 2048, the generator */
} DSAm modulus;

typedef struct _DSAPubkey {
    WORD16 y[128]; /* maximum 2048 bits public key */
} DSAPubkey;

typedef struct _DSAPrivkey {
    WORD16 salt[2] /* rndm data for covering purposes */
    WORD16 x[10]; /* fixed private key size, 160 bits */
} DSAPrivkey;

```

25/45

FIG-28 DSA DIGITAL SIGNATURE OBJECT DEFINITION

```

typedef unsigned short WORD16;      /* 16 bits of data */
typedef void *VPTR;                /* pointer to any object */

typedef struct _signblock {
    WORD16 r[10]; /*  $r = (g^k \bmod p) \bmod q$  */
    WORD16 s[10]; /*  $s = (k^{-1} (H(m) + x r)) \bmod q$  */
} signature_block;

```

FIG-29

```

typedef unsigned short WORD16;      /* 16 bits of data */

typedef struct _seedkey {
    WORD32 counter; /* 32 bit counter to verify prime number */
    WORD16 seed[10]; /* 160 bit random data to generate a prime p and q */
} seedkey;

```

26/45

FIG-30 KEY CACHE REGISTER DATA TYPE DEFINITION

```
typedef unsigned short kcr; /* 16 bits only */
```

FIG-31

```
typedef unsigned short WORD16;
typedef unsigned short BOOL;

typedef struct _crypto_cntxt {
    unsigned short config; /* use one of the constants defined in the document */
    kcr 5 key; /* titled, Cryptic Command Interface Specification */
    WORD16 iv[4]; /* KCR ID for secret key */
} crypto_cntxt; /* 64 bit for IV, feed-back register */
```


27/45

FIG-32 ONE-WAY HASH CONTEXT STORE

```

typedef unsigned char BYTE;
typedef unsigned short WORD16;
typedef unsigned long WORD32;

typedef struct {
    WORD32 count[2]; /* number of bits, module 2^64 (lsb first) */
    WORD32 state[4]; /* state (ABCD) */
    UINT16 buffer[32]; /* input buffer */
} MD5_CTX;

typedef struct {
    WORD32 count[2]; /* 64-bit bit count */
    WORD32 digest[5]; /* Message digest */
    WORD32 data[16]; /* SHS data buffer */
} SHS_INFO;

typedef struct _hash_cntxt {
    WORD16 algorithm; /* Either CGX_MD5_A or CGX_SHS_A */
    BYTE *digest; /* Pointer to message digest [either
                  * 128-bits for MD5 or 160-bits for SHS.
                  */

    /* The following buffer stores the algorithm's state
     * information. (96 bytes)
     */
    union {
        MD5_CTXmd5;
        SHS_INFOshs;
        state
    } hash_cntxt;

```

28/45

FIG-33 EXAMPLE OF CGX WRAP CODE AND COMMAND INTERFACE

```

#include "cgx.h"

#define cgx_encrypt(kb, datain_pg, datain, dataout_pg, dataout, size, cryptoblock) {\
    kb->cb->cmd      = CGX_ENCRYPT; \
    kb->cb->argument[0] = (VPTR)cryptoblock; \
    kb->cb->argument[1] = (VPTR)datain_pg; \
    kb->cb->argument[2] = (VPTR)datain; \
    kb->cb->argument[3] = (VPTR)size; \
    kb->cb->argument[4] = (VPTR)dataout_pg; \
    kb->cb->argument[5] = (VPTR)dataout; \
    cgx_transfer_secure_kernel(kb); }

boolean
encrypt_packet_ecb(unsigned char *data, unsigned size, kcr key, kernelblock *kb)
{
    crypto_cntxt *cb;

    cb = (crypto_cntxt *)malloc(sizeof(crypto_cntxt));

    cb->config = CGX_DES_ECB_M;
    cb->iv      = (WORD16 *)NULL;
    cb->key1     = key;

    cgx_encrypt(kb, 0, data, 0, data, size, cb); // request CGX Kernel to perform
                                                // encryption of the data packet, copy
                                                // of result is written back over src.

    free((unsigned char *)cb);
    /* check the result of the operation */
    if ( kb->sb->status == CGX_SUCCESS )
        return TRUE;
    else
        return FALSE;
}

```

29/45

FIG-34 CGX OVERLAY TABLE DEFINITION

```
typedef unsigned short WORD16;          /* 16 bit data */
typedef void      (*CGX_FUNC) (void); /* function ptr */

typedef struct _cgx_overlay_tuple {
    CGX_FUNC cgxf; /* CGX overlay operation */
    WORD16 control; /* control variable: preempt */
} cgx_overlay_tuple;
```

30/45

FIG-35 KCS OBJECT DEFINITION

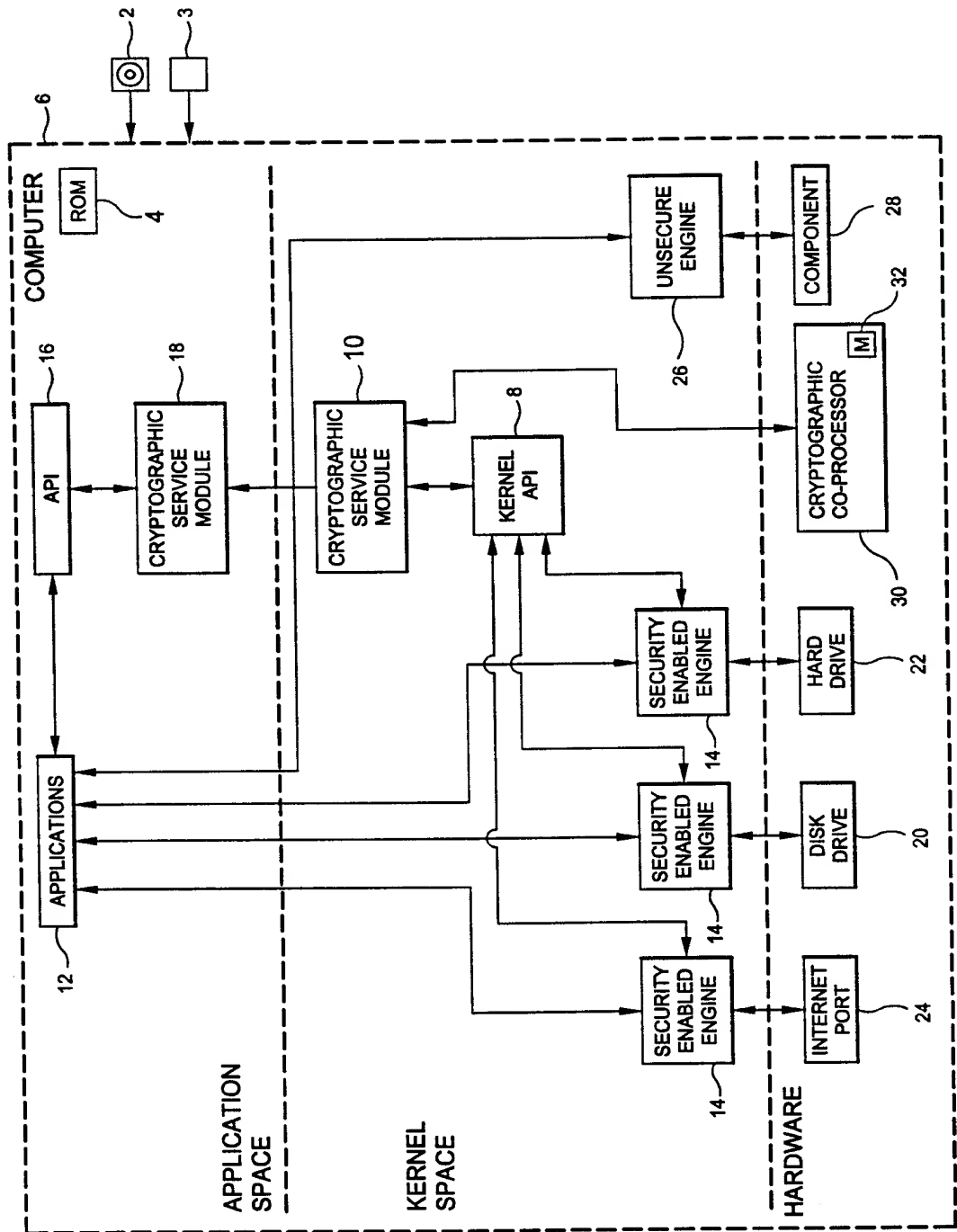
```

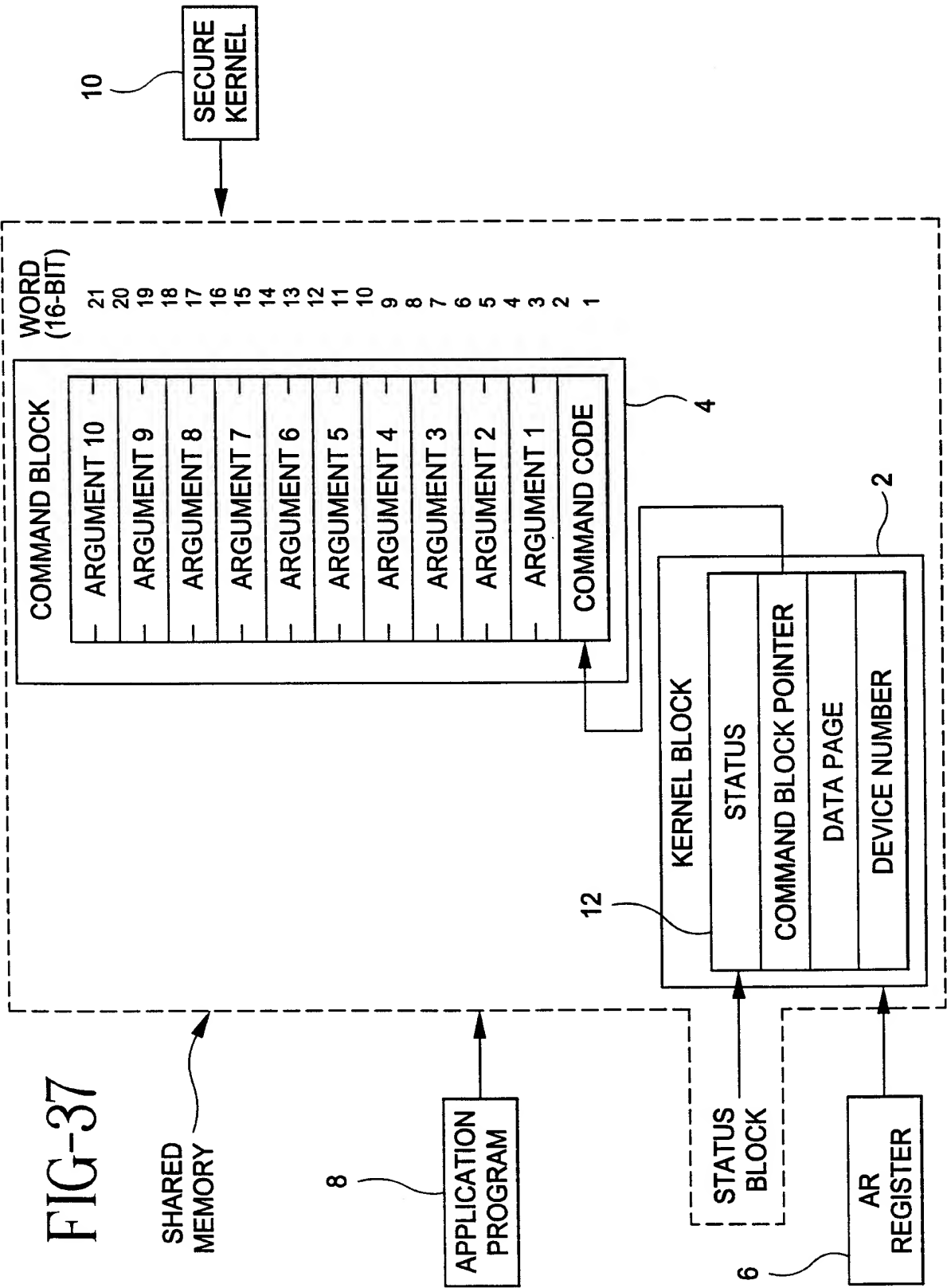
#define CGX_KCS_VERSION 0x0001
#define CGX_KCS_RESERVED_COUNT2
typedef struct _kcs {
    UINT16 version;
    UINT16 reserved[CGX_KCS_RESERVED_COUNT];
    UINT16 rng_alarm_count_threshold;
    UINT16 rng_enable;
    UINT16 kernel_blocked;
    UINT16 kernel_dsp_dma;
    UINT16 kernel_dsp_dma_ifc;
    UINT16 dma_max_dwords;
    UINT16 kernel_dsp_hashenc;
    UINT16 kernel_dsp_hashenc_ifc;
    UINT16 kernel_host_hashenc;
    UINT16 kernel_host_hashenc_ifc;
    UINT16 hashenc_iv;
    UINT16 hashenc_cntl;
    UINT16 fips140_1;
    UINT16 flipsha;
    UINT16 flipgxy;
    UINT16 rng_control;
    UINT16 rng_config;
} kcs;

/* version of KCS string */
/* Reserved fields */
/* RNG Alarm Count Threshold */
/* Enable RNG Clock */
/* address of semaphore */
/* address of semaphore */
/* IFC for knl DMA release */
/* Max dwords knl DMA trans */
/* address of semaphore */
/* IFC for knl HASHENC rls */
/* enable bus grant/ack */
/* IFC for knl HASHENC rls */
/* IV write disable cntl */
/* Cntl who owns the HASHENC */
/* Enable fips140.1 mode */
/* Control flips of SHA md */
/* Control flips g^xy of DH */
/* Used to control the RNG */
/* Used to configure the RNG */

```

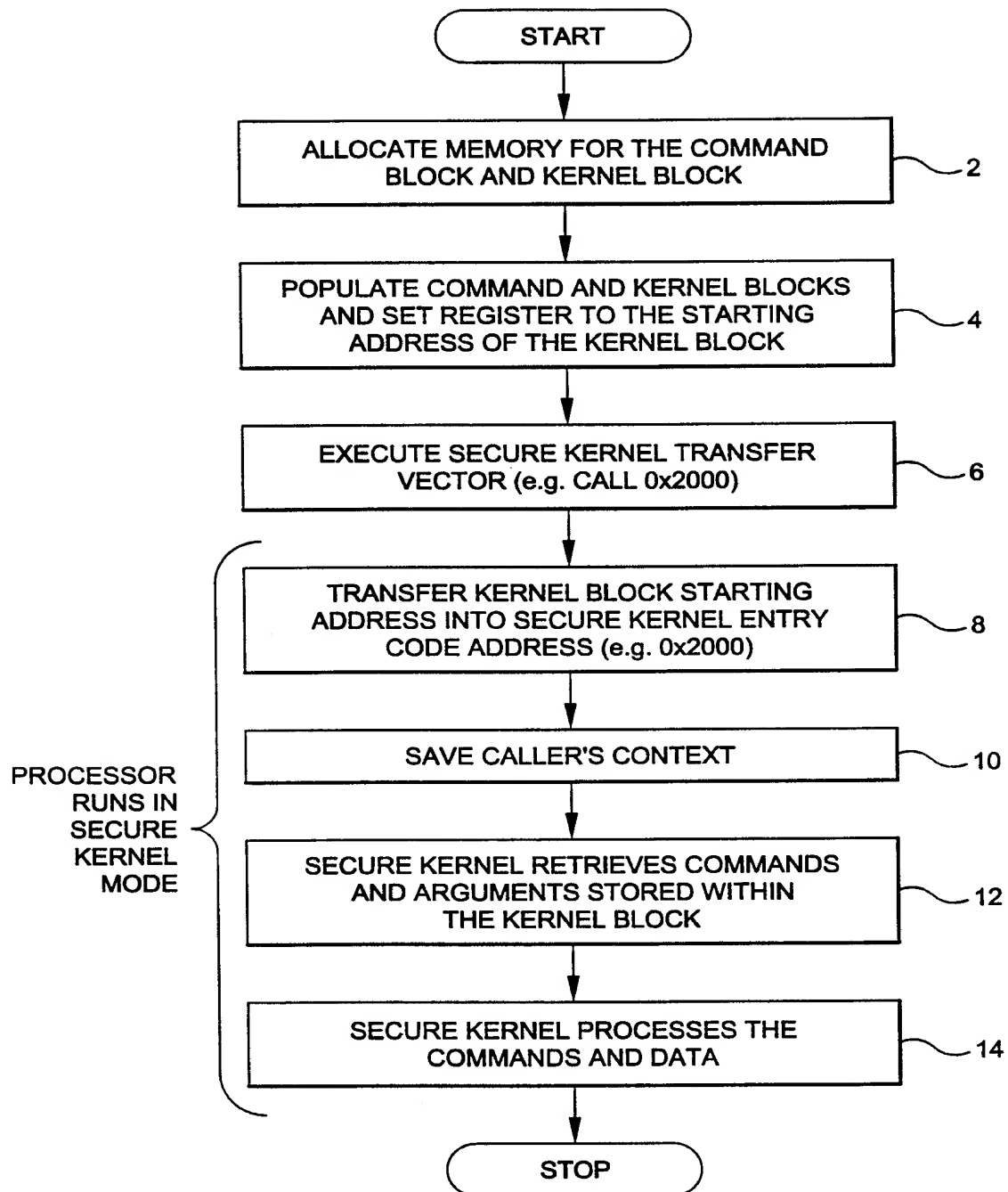
FIG-36





33/45

FIG-38



34/45

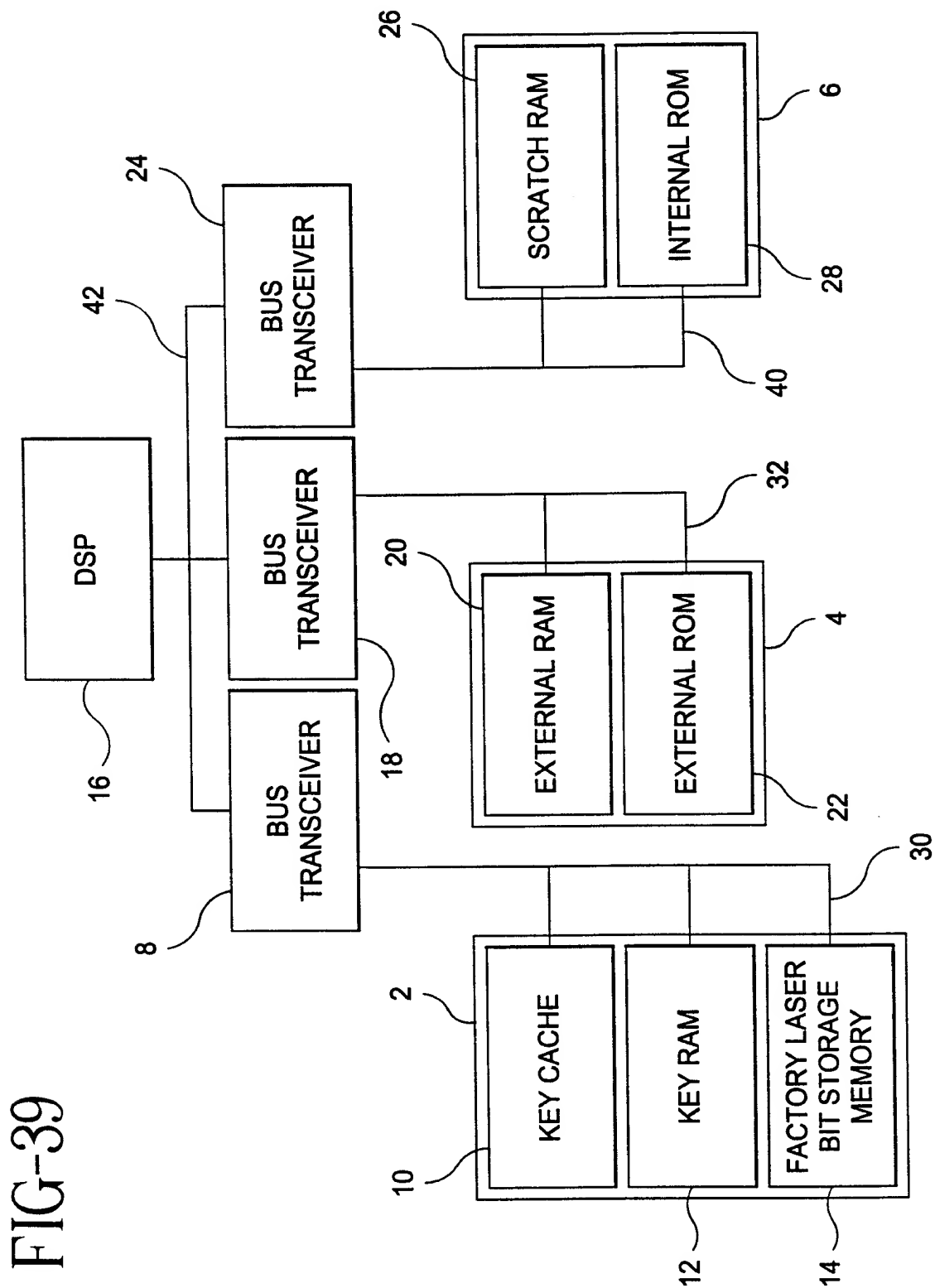
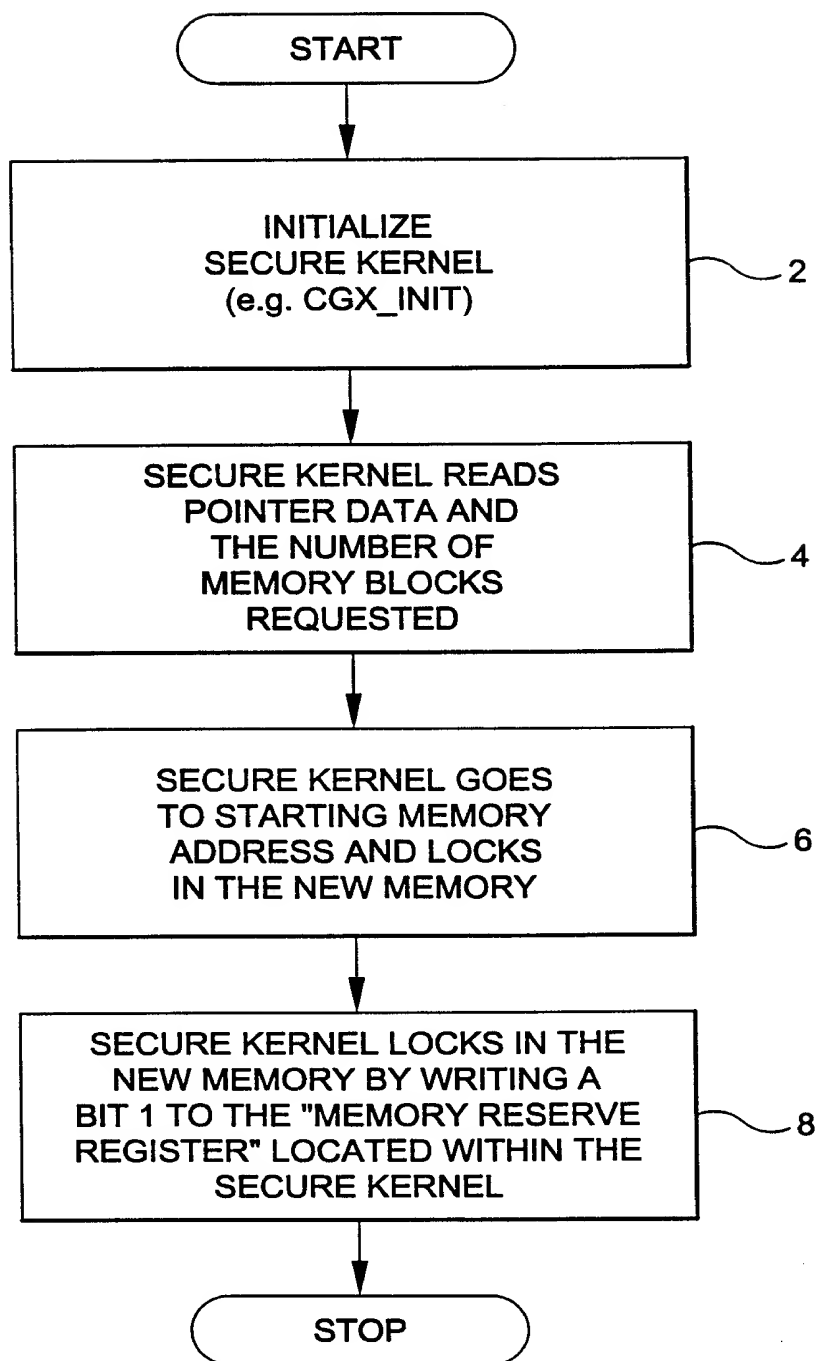


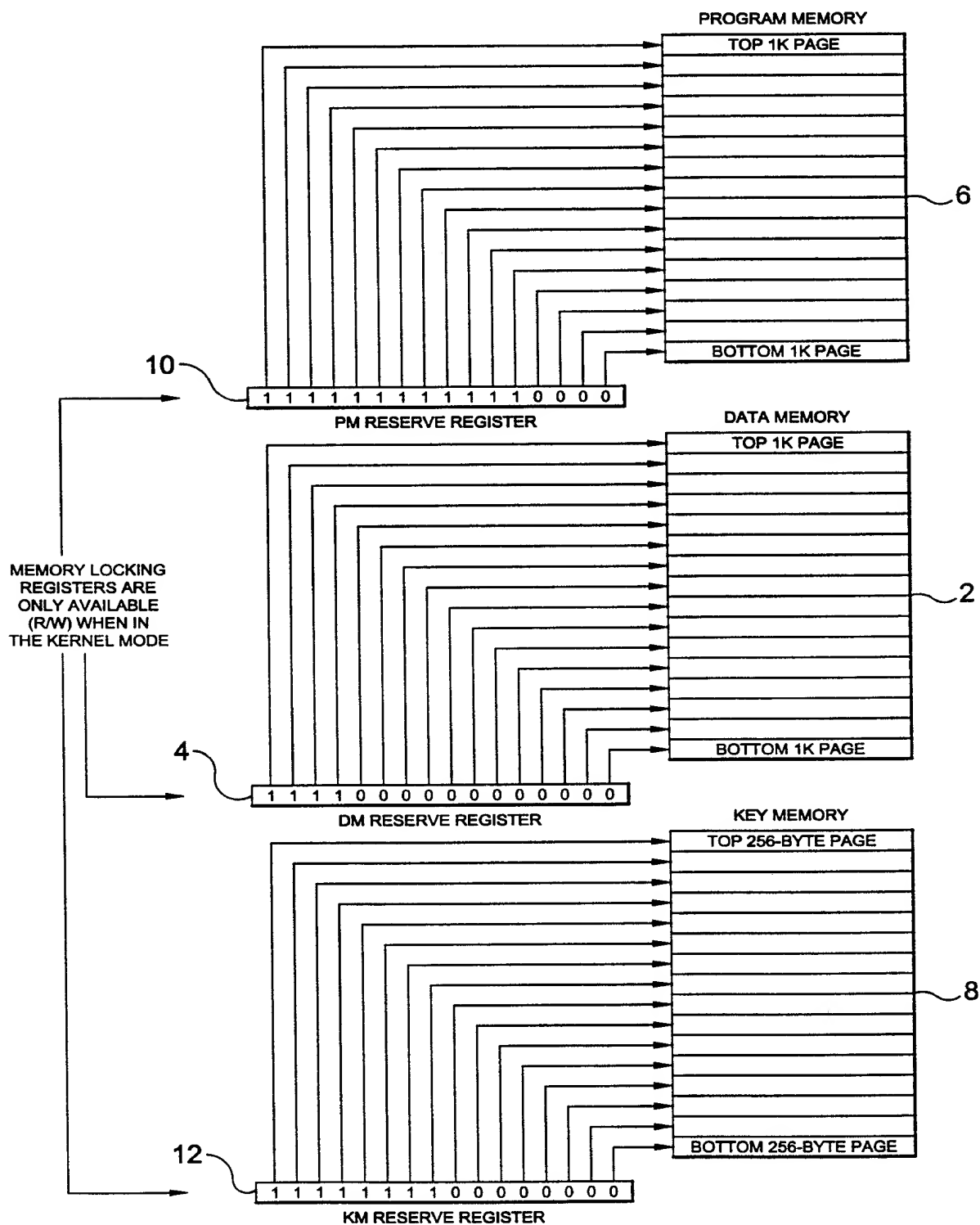
FIG-39

35/45

FIG-40

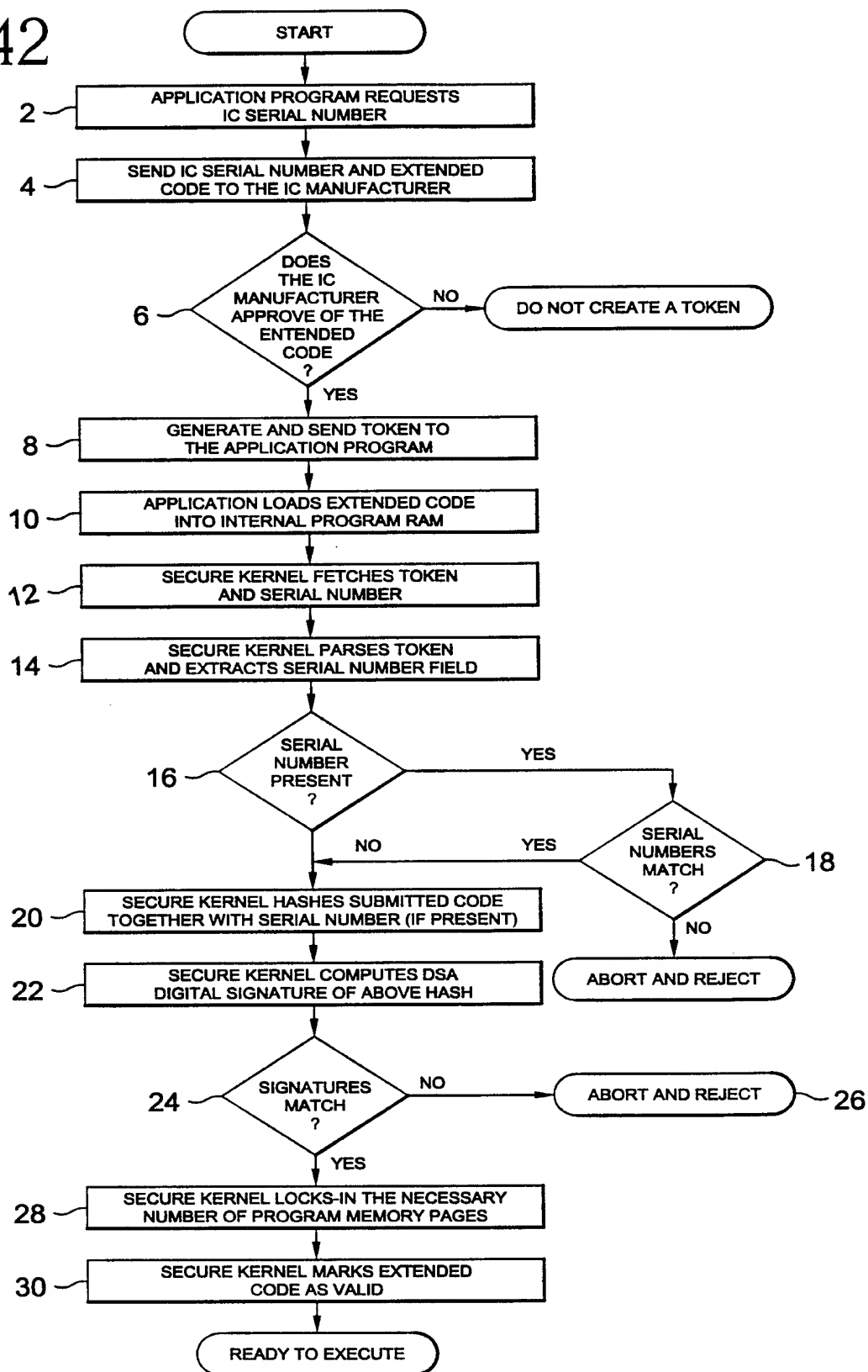


36/45

FIG-41 MEMORY LOCKING WITH xM RESERVE REGISTERS

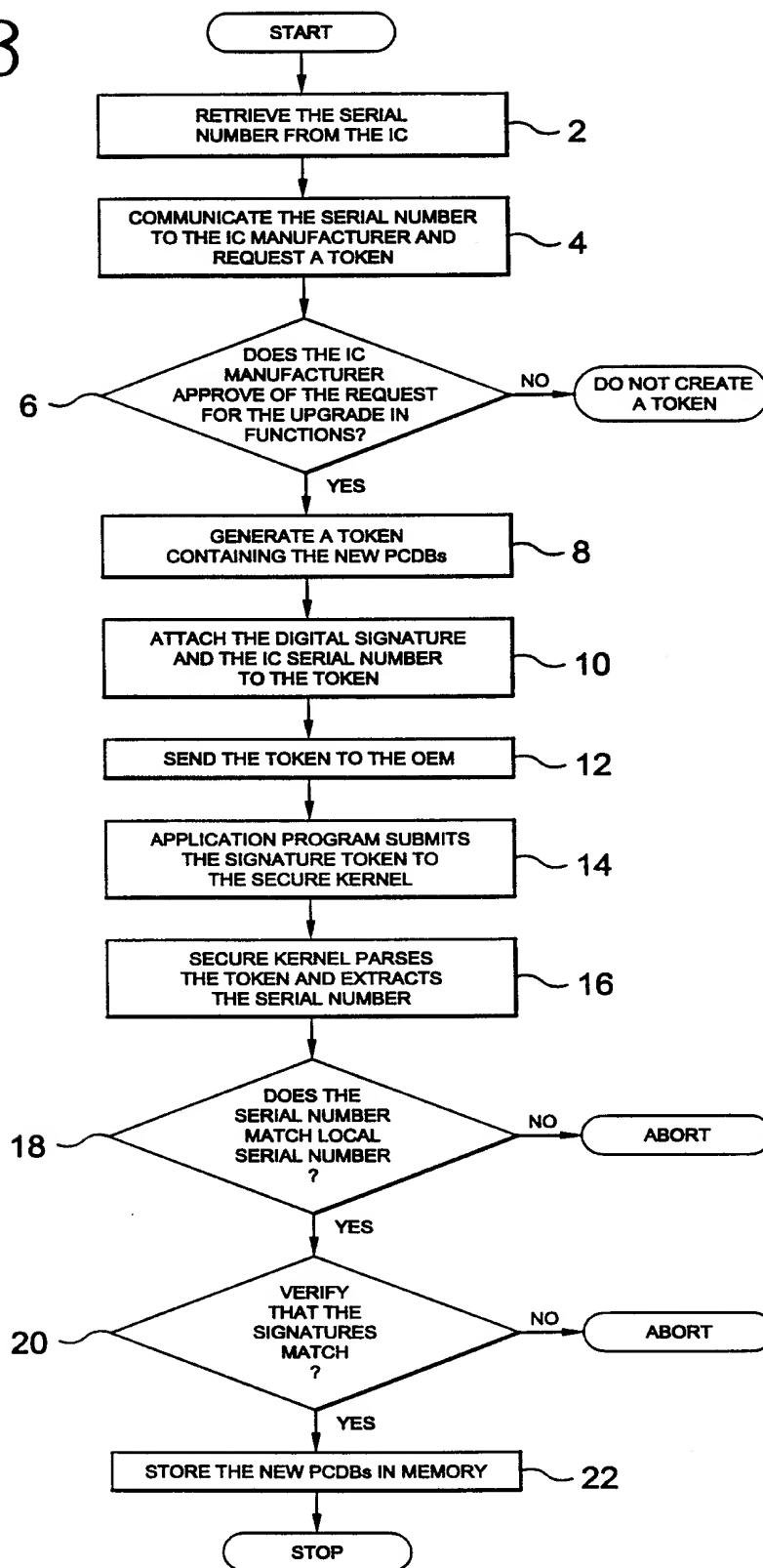
37/45

FIG-42



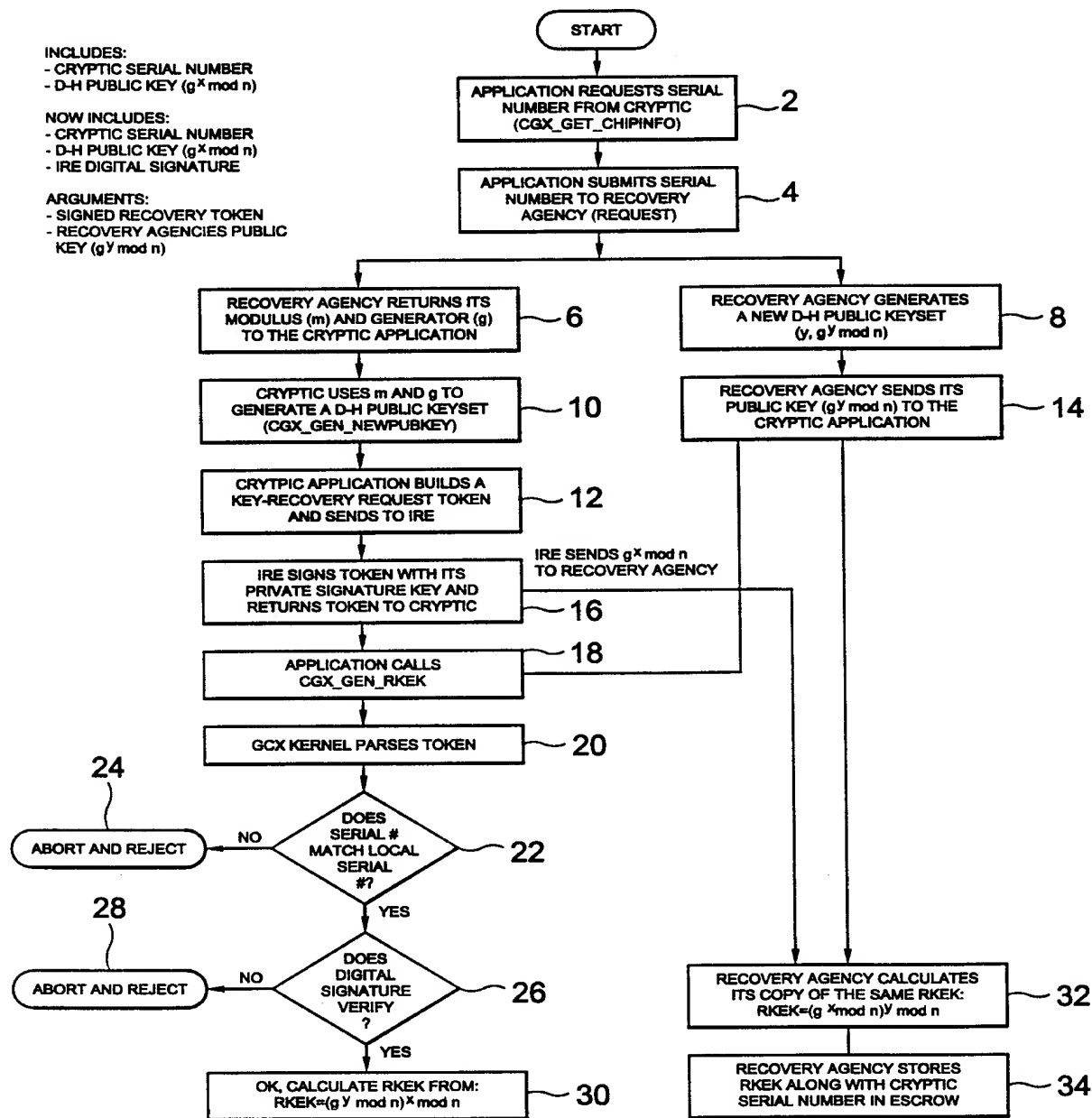
38/45

FIG-43

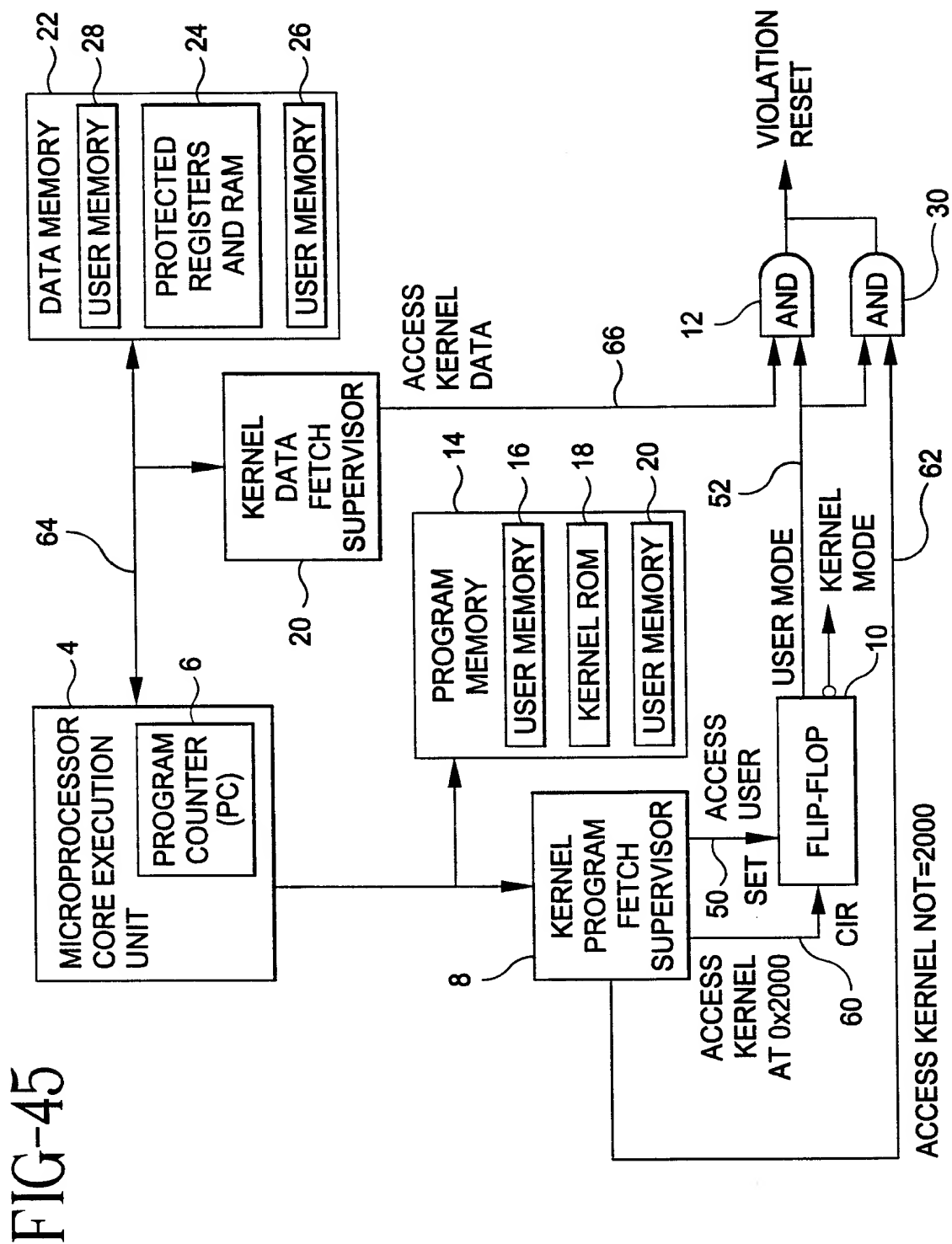


39/45

FIG-44 KEY RECOVERY FLOW DIAGRAM

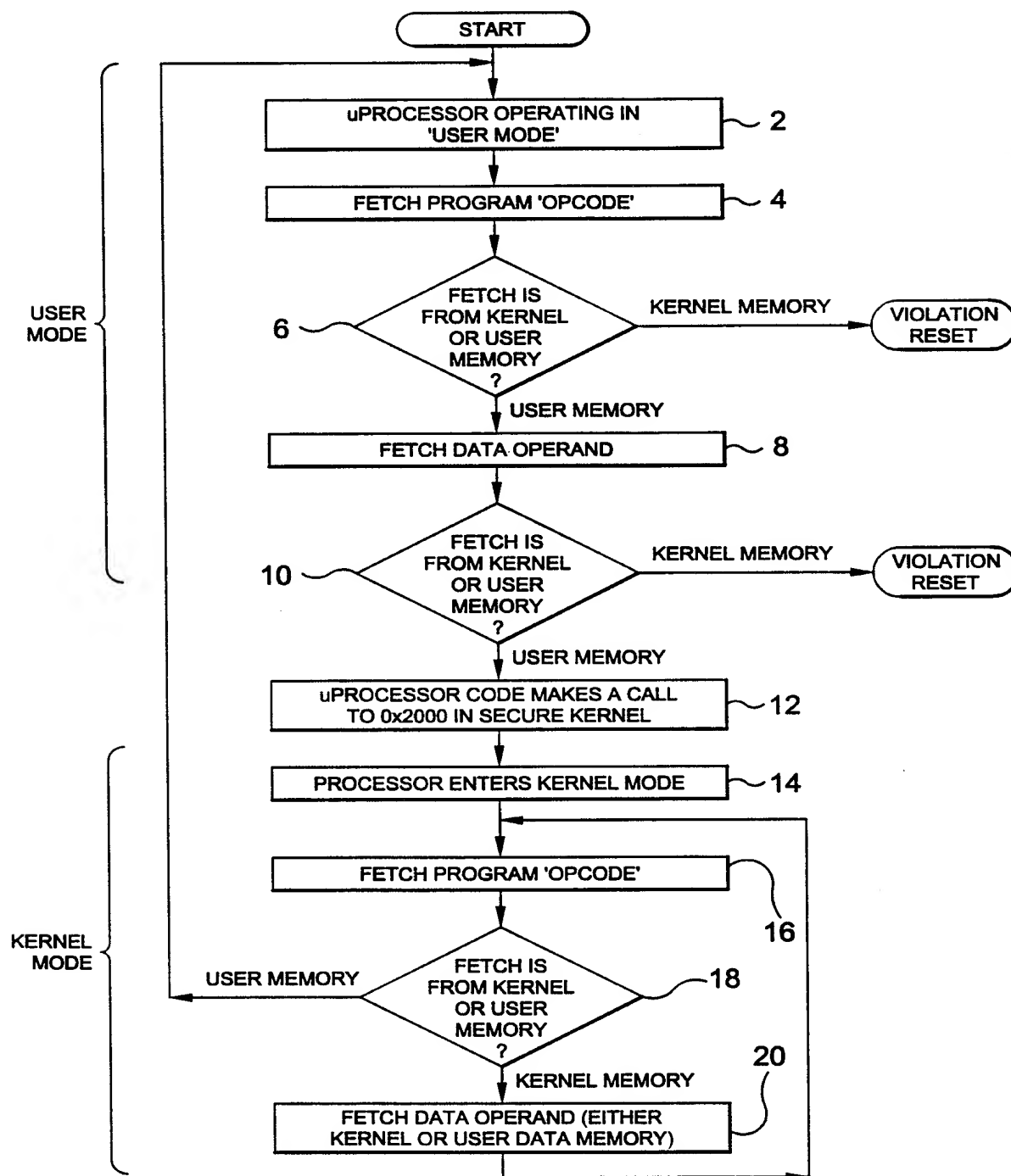


40/45



41/45

FIG-46



42/45

FIG-47

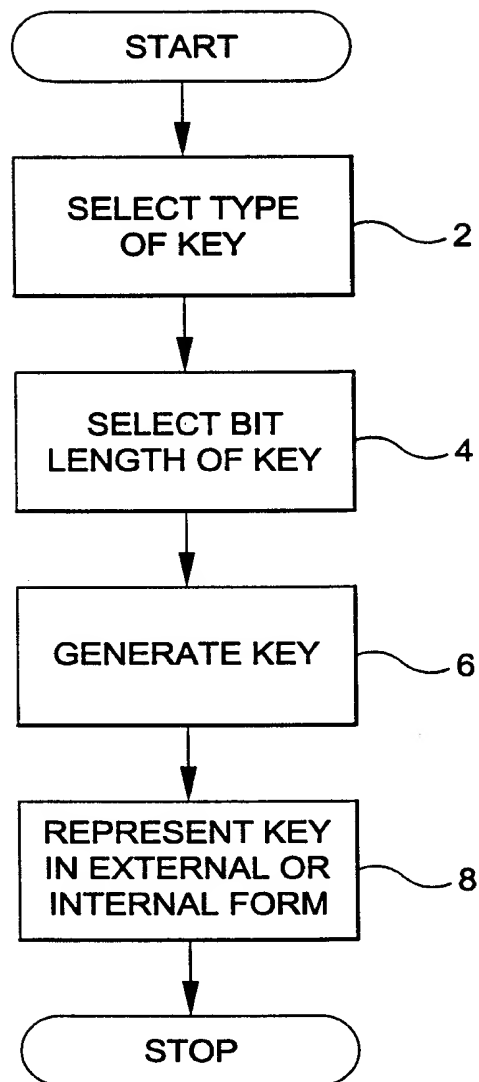


FIG-48

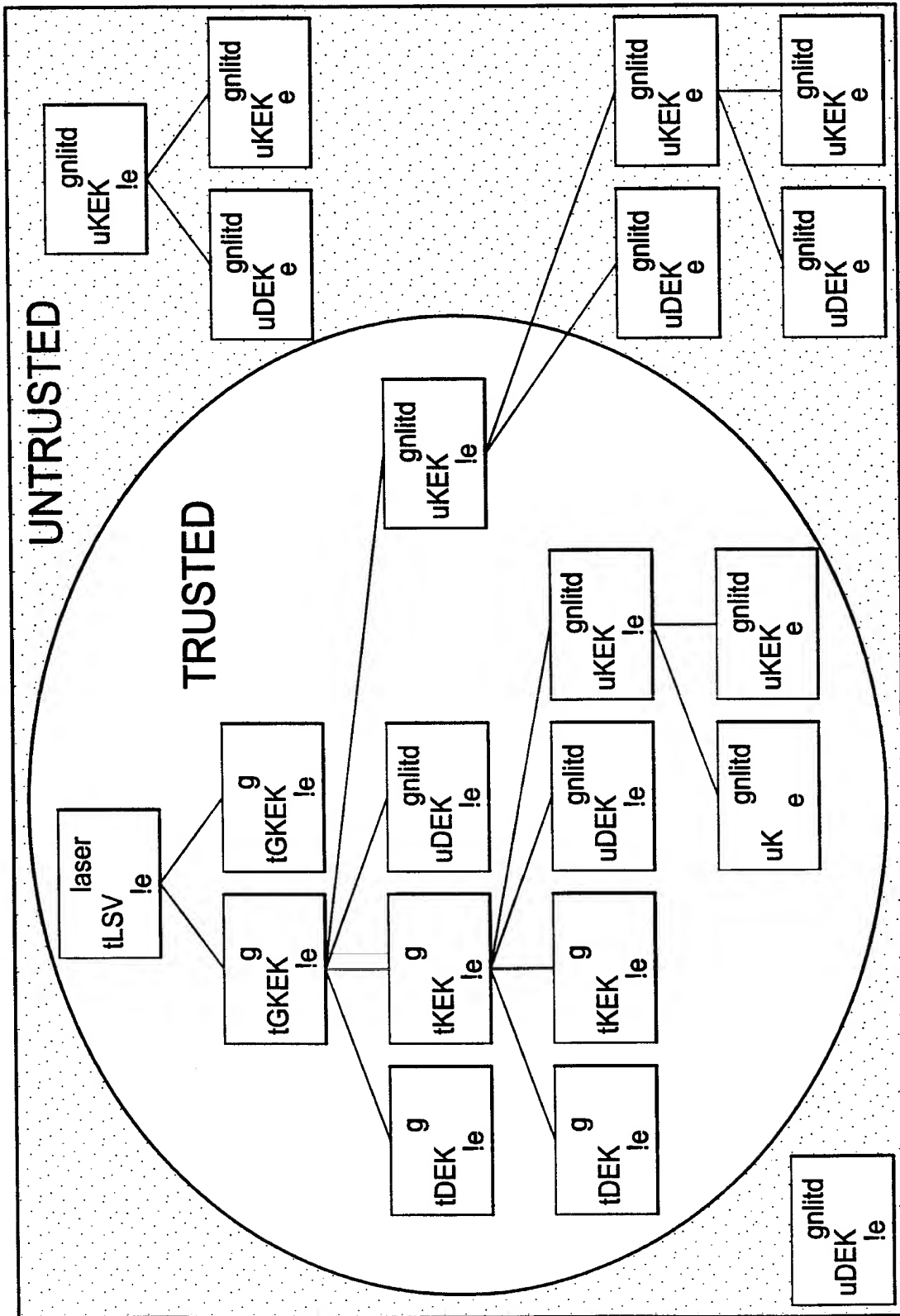


FIG-49

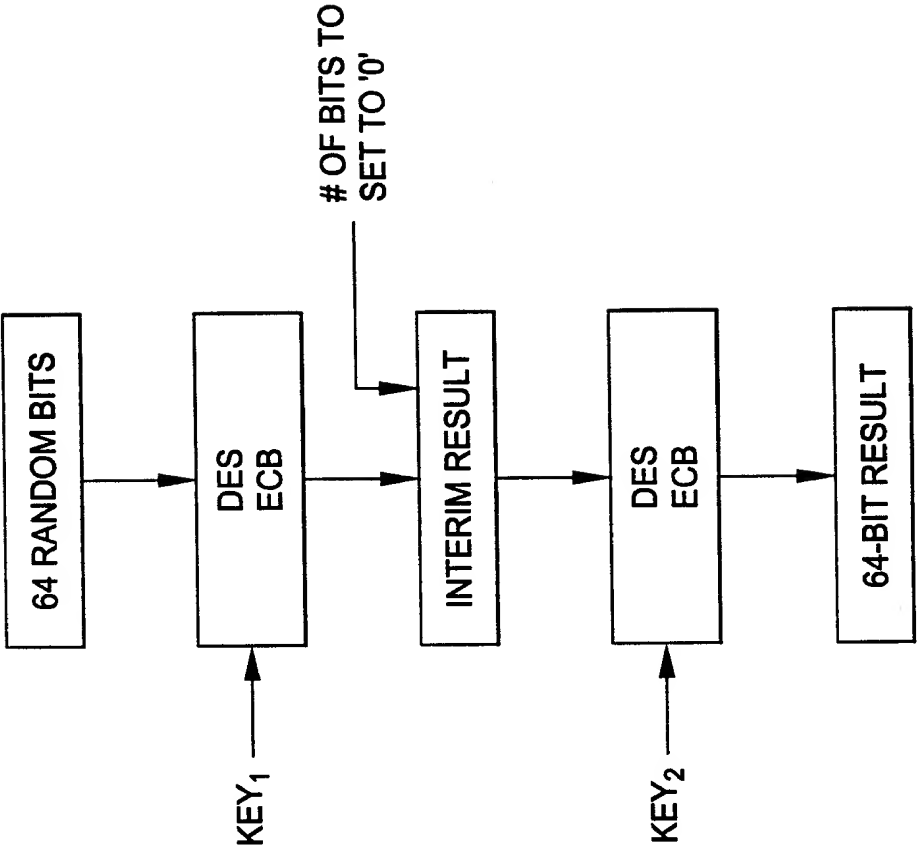


FIG-50

